Object-Oriented Programming With ANSI-C

Axel Schreiner

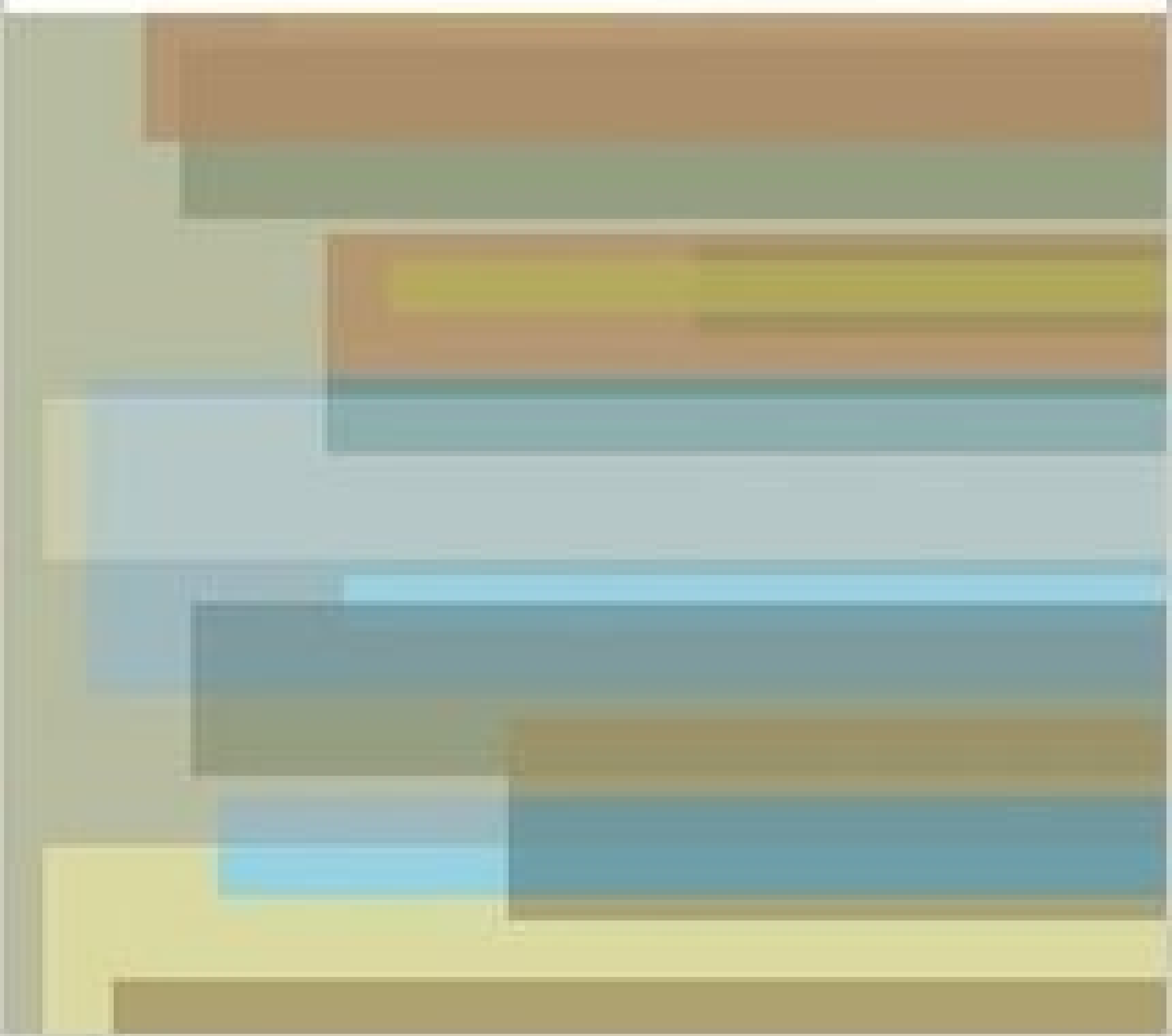


Table of Contents

ANSI C 面向对象编程 1.1
序言 1.2
第一章 抽象数据类型——信息隐藏 1.3
第二章 动态链接和泛函数 1.4
第三章 编程的悟性——算术表达式 1.5
第四章 继承——代码重用和改进 1.6
第五章 编程经验——符号表 1.7

ANSI C 面向对象编程（前五章）

> 原书：Object-Oriented Programming With ANSI-C

序言

> 来源：http://blog.csdn.net/besidemyself/article/details/6376405
>
> 译者：besidemyself

没有能解决所有问题的编程技术。

没有只产生正确结果的编程语言。

没有从头开始每个项目的程序员。

面向对象程序设计几乎是当今包治百病的——虽然它已经发展了超过10年之久。作为一种核心语言，一些技术专家对它的研究已经付出很多，从而形成了很好的编程规则，这些规则我们一直引以为鉴了长达20年之久。C++（Eiffel，Oberon-2，Smalltalk… 由你选择）是一种新的编程语言，因为它是面向对象的——尽管你不需要使用它或许你不想用（或不知道怎么用），但是你却能够使用平凡的ANSI-C （标准化）像使用这些面向对象语言一样的方便。在项目当中，仅仅面向对象语言允许代码重用——虽然一些子程序的思想就像计算机的发展历史一样久远，好的程序员却总是会利用工具箱或库来使用这些子程序。

这本书的本意并不是赞扬面向对象程序或批判老的编程规则。我们只是简单的使用ANSI-C来发掘面向对象语言是怎么实现的，技巧方法是什么，为什么它能够帮助我们解决比较大的问题，和我们怎么能利用普遍性方法和编程来更早的捕获错误。沿着这么一个宗旨，我们会邂逅一些行话—类，继承，实例，链接，方法，对象，多态等等—不过我们剥去其神奇的外衣，看如何转化为一种我们一直都知道和做过的事情。

我很乐意的发掘ANSI-C（标准化）是一种全面的面向对象语言。为了能够分享这种乐趣，在你开始之前，你需要适当的对ANSI-C 有一定的流利程度—熟悉结构体，指针，原型，并且掌握函数指针是必须的。贯穿全文，你会遇见所有的新说法 — 依照Orwell和Webster的话：“设计是为了缩小思想的广度”—并且我会尽力证明它是怎样的结合所有的这些你一直想连续使用的好的方法，结果，你很可能成为一个精通ANSI-C的程序员。

前六章建立ANSI-C面向对象程序设计的基础。我们将以一个隐藏抽象数据结构的很精确的信息开始，然后通过结构体的扩展，基于动态链接和代码继承的方式来增加功能属性。最终把他们放到一起构成一个可继承的类使得代码更加容易维护。

编程需要规则。好的程序遵循很多规则，大量的规则，标准，自我防御的方式往往能使事情做对做好。程序员要学会使用工具。优秀的程序员编写工具去处理日常一些编程例程。ANSI-C 面向对象程序要求有一定数量固定的代码 — 即名称改变，数据结构没有变。因此，在第七章我们建立了一个小的预处理器创建一个必须的样板，它更像另外一种新的面向对象语言（也许是yanoodl）但是它并不被如此看待，OOC（面向对象程序设计ANSI-C）拓新而出，更让我们专注于在解决问题上使用创造性方面的一种新技巧。OOC有很强的可塑性。我们设计出它，理解它并且可改变它，能够依照我们的意愿去写ANSI-C代码。

接下来的章节中将精炼我们的技术。在第八章节，我们增加了动态类型检查预先的捕获异常。第九章中我们编排了自动初始化机制防止其它缺陷类的产生。第十章，我们介绍了多态以及怎样相互协作达到简化的目的。例如：产生标准主程序的日常事务。更多的章节将与使用类的方法，存储，和加载结构体数据相关联，这些结构体数据遵循一致性策略。并且通过嵌套异常处理器系统来做到一致性错误恢复自愈。

最终，在最后一章节中，我们避开ANSI-C的局限性并且实现了一个鼠标操作的计算器，首先对于 curses 终端，接下来应用于 X Window 系统（如果你想了解什么事curses 和x window 查阅相关资料）。 这个例子清晰的证明了我们使用类和对象所做的设计和实例的优美性，虽然我们必须处理外在的库和类层次特性。

在每个章节的前面都有一个概要说明，在这个概要中我会给出一个纲要，来介绍章节中的主要内容和接下来该做什么，这对于略读此章的读者很有帮助。大部分章节建议做一下练习；但是这并不意味着很正规，因为我们是从零开始建立这样的技术。我已经避免制造和使用庞大的类库，即使一些例子这样做似乎很有利。如果你想更好的理解面向对象程序设计，掌握这项技术和考虑代码设计显得尤为重要；依靠其他人的类库来做开发是稍后小菜一碟的事。

这本书的一个重要的组成部分是附上了代码软盘---- 使用DOS文件系统，包含了单独的SHELL命了来创建所有章节中的代码。软盘上有一个ReadMe 文件----再你生成代码之前先来阅读一下它。编程，就像使用如 diff 的程序，跟踪基类的演变对你会很有好处的。OOC在接下来的章节中会有很好的体现。

这些技术的描述出自我对C++的觉悟，当我需要使用面向对象技术去实现交互式编程语言时。并且意识到在C++中不能铸造一个便携式的实例。我转向我所了解的，即ANSI-C，我能够完全做我必须做的。我已经在教学过程中把这项技术分享给很多人，包括我的工作场所。而且其他人已经使用这些方法很好了完成了他们的工作。这本书就驻留于此，因为对我的脚注却显得尤为暗淡。Brian Kernighan ，我的出版社，Hans-Joachim Niclas和John等等，并没有鼓励我出版这些笔记（有机会，在适当的时候会从新组织一次），我感谢他们以及帮助我去继续这本书的人，最后，感谢我的家人——并且，不，面向对象机制不会取代“切片面包”。

Hollage, October 1993

Axel-Tobias Schreiner

第一章 抽象数据类型——信息隐藏

> 来源：http://blog.csdn.net/besidemyself/article/details/6376408
>
> 译者：besidemyself

1.1 数据类型

数据类型是每种编程语言不可或缺的一部分。ANSI-C（标准化C）拥有一些基本数据类型：`int` ，`double` 和 `char` 。有限的数据类型几乎不能满足程序员的要求，所以编程语言会提供一种机制来使得程序员使用这些基本的预定义数据类型构造新的数据类型。一个简单的应用就是构造集合，如数组，结构体和联合体。而指针集，依照C.A.R Hare的话："从这一步起，我们也许永远不会复苏"允许我们描述和操作本质上无限复杂的数据。

什么才是真正的数据类型呢？我们可以发表不同的观点。数据类型是一系列值的集合——`char` (字符型)数据类型拥有256个不同的值，`int` （整型）数据类型拥有更多不同值，他们之间的间隔相等，表现形式多少有点像数学中的自然数或整数，而 `double` (浮点数据类型)类型拥有更多的可能的值，类似数学中的带小数部分的数。

有选择性地，我们能够定义一种数据类型作为一个值的集合加上一系列操作来做一些事情。典型地，定义的这些值计算机能够表示，这样的操作过程能够翻译成机器指令。在这方面，`int` 型在标准C语言中做的并不是很好。这些数据的集合的值的范围可能随着不同机器而不同，操作方式就像算术中的右移操作，表现形式可能不尽相同。

太复杂的例子往往不能得到有效的说明。我们可以典型地把一个线性列表中的元素定义成一个结构体如下：

```
typedef  struct node {
    struct node *Next;
    …information…
}node ;
```

并且，对于这个列表的操作，我们指定列表的头，如下：

```
node * head (node * elt , const  node *  tail);
```

然而，这样的应用是非常冗余的，好的编程规则指示我们隐藏数据项的表示，仅仅声明操作方法。

1.2 抽象数据类型

如果我们不把对这个数据类型的表示呈现给用户，则我们称这个数据类型为抽象数据类型。从理论的角度，要求我们通过包含可能性操作的数学表达式中指定数据类型的属性。例如，我们从队列中删除一个我们先前增加的元素，并且可以从队列中以相同的次序检索我们增加的元素。

抽象数据类型为程序员提供了很强的便利性。因为表达式不是定义的一部分。我们可以自由的选择更简单，更有效的方式去实现。如果我们能够正确的分离出必要的信息，那么对数据类型的使用和实现将完全的独立开来。

抽象数据类型满足了“信息隐藏”和“分而治之”的良好编程规则。例如数据项表达式—— 只给需要知道的人提供，给抽象数据类型的实现者，而不是用户。通过使用抽象数据类型，我们可以清晰的分离出实现者和使用者的不同任务。并且可很好的把一个大的系统分解成各个小的模块。

1.3 举例——集合

由此，我们怎样的实现一个抽象数据类型呢？我们以对一个集合中的元素操作为例，使用操作方法， `add` （增加）， `find` （查找），和 `drop` （删除）。这些方法均用于集合和集合中的元素。 `add` 方法向集合中添加一个元素，并返回要添加的元素， `find` 方法从集合中查找指定的元素，可被用于实现来判断一个指定的元素是否在集合中， `drop` 方法从集合中删除一个元素。

使用这种方式，即可看到 `set` (集合)是一个抽象的数据类型。现在声明一下我们想要做的，以一个头文件 `Set.h` 开始：

```
#ifndef __USR_SET_H__
#define __USR_SET_H__
extern const void * Object;
void* add(void* set,const void* element);
void* find(const void* set,const void *element);
void* drop(void *set,const void * element);
int contains(const void* set,const void* element);
#endif
```

前两句的作用可使编译器对这段声明加以保护处理。无论头文件 `Set.h` 被包含多少次，C编译器只对这个声明编译一次。这样的声明头文件的方法是很标准的，GNU C 预处理器能够识别，而且当保护符号（如上的 `__USR_SET_H__` ） 被定义，则保证不会再进入保护区声明的代码。

`Set.h` 很完整，但它真的很有用吗？ 我们几乎不可能发觉和想象出它的不足： `Set` （集合）理所当然代表一个实例，我们可以使用这个实例来做很多事情。 `Add()` 方法传递一个元素，并把它添加到集合，并且返回所添加的元素或集合中已经存在的元素； `find()` 在指定的集合中查找元素，并且返回找到的元素，若没有找到，则返回 `NULL` （空）； `drop()` 定位一个元素，把这个元素从集合中删除，并且返回删除的元素； `contains()` 的本质就是把 `find()` 方法所查找的结果转换为“真”值。

通用指针类型 `void*` 的应用贯穿全文。一方面它使得我们想发现集合到底是什么东西成为不可能。但是另一方面它允许我们向如 `add()` 和其他方法中传递任意类型的数据。并不是每件事都会拥有像集合和集合中的元素一样的表现形式——在信息隐藏的乐趣中我们牺牲了类型的安全性。然而，我们可以在第八章看到这样的应用会非常之安全。

1.4 内存管理

也许我们已经瞥见了某些东西：怎样的获得一个集合呢？`Set`（集合）是一个指针，并不是被 `typedef` 关键字定义的类型；因此我们不能把 `Set` 定义成一个局部或全局的类型。相反的我们只是使用指针来引用集合和集合中的元素，并且建立一个文件 `new.h` ，并声明如下：

```
void * new (const void * type, ...);
void delete (void * item);
```

就像 `Set.h` 文件一样的做法，文件被预处理器符号 `NEW_H` 保护起来。以后只列出感兴趣的部分，所有的源代码和所有实例的代码均能在光碟中找到。

`new()` 接收一个像 `Set` 的描述符,传递更多可能的参数用于初始化操作，返回一个指向新数据项的携带描述符信息的指针。 `delete()` 接受一个由 `new()` 所原先产生的指针，并回收关联的资源。

`new()` 和 `delete()` 可看成类似于标准C函数 `calloc()` 和 `free()` 。如果的确是，描述符得能够指示出至少需要申请多大的内存空间。

1.5 （`Object`）对象

如果我们想搜集在集合感兴趣的东西，我们需要另外一个抽象数据类型 `Object` ,在头文件 `Object.h` 中有如下描述：

```
extern const void * Object; /* new(Object); */
int differ (const void * a, const void * b);
```

`differ()` 是用来做对象比较的：即若两个对象不相等则返回眞，否则返回假。这样的描述为C语言函数 `strcmp()` 留有余地：因为在某些比较中我们也许选择返回一个整数或负数的值来指示排列的次序（正序或倒叙）。

现实生活的对象需要更多的功能去做有用的事情。此刻，我们约束我们自己只对集合中的成员（必须品）操作而已。如果我们建立一个更大的类库，我们将看到所谓的集合 — 实际，包括其他所有东西 — 均是一个对象。从这个观点出发，很多的对象包含的功能其实都是无条件存在的。

1.6 应用

包含头文件，和库信息，抽象数据类型，我们能够写一个 `main.c` 的应用程序如下所示：

```
#include <stdio.h>
#include "New.h"
#include "Object.h"
#include "Set.h"

int main()
{
	void* s=new(Set);
	void* a=add(s,new(Object));
	void* b=add(s,new(Object));
	void* c=new(Object);

	if(contains(s,a)&&contains(s,b)){
		puts("ok");
	}

	if(contains(s,c)){
		puts("contains?");
	}

	if(differ(a,add(s,a))){
		puts("differ?");
	}

	if(contains(s,drop(s,a))){
		puts("drop?");
	}

	delete(drop(s,a));
	delete(drop(s,a));

	return 0;
}
```

我们创建了一个结合并给集合中添加了两个新建的对象。如果不出意外，我们可以发现对象会在集合中，并且我们不会再发现其他的新对象。程序的运行会简单得打印出ok 。

对 `differ()` 的调用会证明出这样的语义：数学上的集合只包含集合 `a` 的一份拷贝；对一个元素的重复添加必须返回已经加入的对象，因此上述程序的 `differ()` 为 假。相似的，一旦我们从一个结合删除一个元素，它将不会再存在于这个集合中。

从一个集合中删除一个不存在的元素将返回一个空的指针传递给 `delete()` 。现在，我们已经指示出 `free()` 的语义且必须是合情合理可接受的。

1.7 一种实现机制—— `Set` （集合）

`main.c` 会编译成功，但在连接和执行之前，我们必须实现其中的抽象数据类型和内存管理。如果一个对象不存储信息，且每个对象最多属于一个结合，我们可以把每个对象和集合当成，小的，独立的，正整数的值。可在 `heap[]` 中通过数组下标来索引到。如果一个对象（这里的对象为数组元素的地址）是一个集合的成员，则数组元素的值代表这个集合。对象指向包含它的集合。

首先的解决方案是非常简单的，即我们把其他模块与 `Set.c` 相结合。集合对象集有相似的呈现方式。因此，对于 `new()` 无需关注类型描述。它仅仅从数据 `heap[]` 中返回非零元素。

```
void * new (const void * type, ...)
{
    int *p ;    /*&heap[1...]*/
    for(p=heap+1;p<heap+MANY;++p){
        if(!*p){/*若heap 中的某个元素为0，则返回这个指针，并把值设为MANY*/
            break;
        }
    }
    assert(p<heap+MANY);
    *p=MANY;
    return p;
}
```

我们是用 `0` 来标记 `heap[]` 中有效的元素；因此不能返回 `heap[0]` 的引用——如果它是一个集合，而集合的元素可以是索引值为 `0` 的对象。

在一个对象被添加到集合当中之前， 我们让它包含无效的索引 `MANY` ，以便于 `new()` 不会再次返回它，请不能误解 `MANY` 是集合的一个成员。

`new()` 能够使用完内存。这是很多"致命性错误"的其中一个。我们可以简单的使用标准化C语言宏的宏 `assert()` 来标记这些错误。一个更理想的实现方式是至少会打印合理的错误信息或使用用户可重写的错误处理机制的通用功能。这也是我们的目的中，编码技术完整性的一部分。在第13章，我们会介绍一种通用异常处理的技术。

`delete()` 必须得严加防范空指针的传入。通过设置其元素的值为 `0` 来进行 `heap[]` 中元素的回收。

```
void delete (void * _item)
{
    int* item=_item;
    if(item){
        assert(item>heap && item,heap+MANY);
        *item=0;
    }
}
```

我们需要统一的处理通用指针；因此，给每个通用指针的变量的前面加上下划线前缀，然后仅仅使用它初始化指定类型的局部变量。

一个集合被它的对象所表示。集合中的每个元素指向它的集合。如果元素包含 `MANY` ，则它可以被添加到一个集合中。否则它已经属于一个集合的元素了。因为我们不允许一个对象属于多个集合。

```
void* add(void* _set,const void* _element);
{
    int * set=_set;
    const int *element=_element;

    assert(set>heap && set<heap+MANY);
    assert(*set==MANY);
    assert(element>heap&& element<heap+MANY);

    if(*element==MANY){
        *(int*)element=set-heap;
    }
    else{
        assert(*element==set-heap);
    }
    return (void*) element;
}
```

`assert()` 在这里稍微显得逊色：我们只关注在 `heap[]` 内的指针和集合不属于其他部分的集合，等等，数组元素的值应该为 `MANY` 。

其他的功能都是很简单的。 `find()` 只查找元素的值为集合索引的元素。若找到，返回元素，否则返回 `NULL` 。

```
void* find(const void* _set,const void * _element)
{
    const int* set=_set;
    const int* *element=_element;
    assert(set>heap && set<heap+MANY);
    assert(*set==MANY);
    assert(element>heap && element<heap+MANY);
    assert(*element);
    return *element==set-heap?(void*)element:0;
}
```

`contains()` 把 `find()` 的结果转换为真值：

```
int contains(const void* _set,const void* _element)
{
    return find(_set,_element)!=0;
}
```

`drop()` 依赖于 `find()` 的结果，若在集合中查找到，则把此元素的值标记为 `MANY` ，并返回此元素：

```
void* drop(void * _set,const void * _element)
{
    int* element=find(_set,_element);
    if(element){
        *element=MANY;
    }
    return element;
}
```

如果我们深入挖掘，一定会坚持被删除的元素要不包含于其他集合中。在这种情况下，毫无疑问会在 `drop()` 中复制更多 `find()` 的代码。

我们的实现是很非传统的。在实现一个集合时似乎不需要 `differ()` 。我们仍然提供它，因为我们的程序要使用这个函数。

```
int differ (const void * a, const void * b)
{
    return a!=b;
}
```

当数组中对象的索引不同时，这个对象必然是不同的，也就是索引值就能区分它们的不同，但一个简单的指针比较已经足够了。

我们已经做完了——对于这个问题的解决我们还没有使用描述符 `Set` 和 `Object` ，但是不得不定义它以使我们的编译器能通过。

```
const void * Set;
const void * Object;
```

我们在 main() 函数中使用上述指针来创建集合和对象。

1.8 另一种实现——包

不需要改变 `Set.h` 中的接口，我们来改变接口的实现方式。这次使用动态内存分配，使用结构体来表示集合和对象：

```
struct Set{
        unsigned count;
};
struct Object{
        unsigned count;
        struct Set* in;
};
```

`count` 用于跟踪集合中的元素的计数个数。对于一个元素来说，`count` 记录这个元素被集合添加的次数。如果我们想递减 `count` 值，可调用 `drop()` 方法。一旦一个元素的 `count` 值为 `0` ，我们就可以删除它，我们拥有一个包，即，一个集合，集合中的元素拥有一个对 `count` 的引用。

因为我们使用动态内存分配机制去表示集合集和对象集，所以需要初始化 `Set` 和 `Object` 描述符，以便于 `new()` 能够知道需要分配多少内存：

```
static const size_t _Set=sizeof(struct Set);
static const size_t _Object=sizeof(struct Object);

const void * Set=&_Set;
const void * Object=&_Object;
```

`new()` 方法现在更加简单：

```
void * new (const void * type,...)
{
        const size_t size= *(const size_t*)type;
        void* p=calloc(1,size);
        assert(p);
        return p;
}
```

`delete()` 可直接把参数传递给 `free()` ——标准化C语言中 一个空的指针可以传进 `free()` 。如下：（如意调用）

```
void delete (void * _item)
{
        free(_item);
}
```

`add()` 方法多多少少对它的指针自变量比较信任。它会增加元素的引用计数和集合的引用计数。

```
void* add(void* _set,const void* _element)
{
        struct Set *set=_set;
        struct Object* element=(void*)_element;

        assert(set);
        assert(element);

        if(!element->in){
                element->in=set;
        }
        else{
                assert(element->in==set);
        }

        ++element->count;
        ++set->count;

        return element;
}
```

`find()` 方法仍然会检查，一个元素是否指向一个适当的集合：

```
void* find(const void* _set,const void * _element)
{
        const struct Object* element=_element;

        assert(element);

        return element->in==_set?(void*)element:0;
}
```

`contains()` 方法基于 `find()` 方法来实现，仍然保持不变。

若 `drop()` 在集合中找到它要操作的元素，它将递减元素的引用计数和元素在集合中的计数。如果引用计数减为0，这个元素即被从集合中删除：

```
void* drop(void * _set,const void * _element)
{
        struct Set* set=_set;
        struct Object* element=find(set,_element);

        if(element){
                if(--element->count==0){
                        element->in=0;
                }
                --set->count;
        }
        return element;
}
```

现在我们可以提供一个新的方法，用来获取集合中的元素个数：

```
unsigned count(const void* _set)
{
        const struct Set* set=_set;

        assert(set);
        return set->count;
}
```

当然啦，直接让程序通过读 `对象.count` 显得比较简单，但是我会坚持不去披露集这样的实现。与应用程序重写临界值的危险性相比上述功能的调用的开销是可忽视的。

包的表现与集合是不同的。一个元素可被添加多次；当一个元素的删除次数等于其被添加的次数时，这个元素被从集合中删除， `contains()` 方法仍然能够找着它。测试程序的运行结果如下：

```
ok
drop?
```

1.9 总结

对于抽象数据类型，我们完全隐藏了其实现的细节，例如应用程序代码中数据项的描述。

程序代码只访问头文件，在头文件中描述符指针表示数据类型，对数据类型的操作作为一种方法被声明，此方法接收和返回通用指针。

描述符指针被传进通用方法 `new()` 中去获得一个指向数据项的指针，这个指针被传进通用方法 `delete()` 中去回收关联的资源。

通常情况，每个抽象数据类型被在单独的源文件中实现。理想情况下，它不对其他数据类型描述。这个描述符指针正常情况下至少指向一个固定大小的值来指示需要的数据项空间大小。

1.10 练习

略。

第二章 动态链接和泛函数

> 来源：http://blog.csdn.net/besidemyself/article/details/6387915
>
> 译者：besidemyself

2.1 构造器和析构器

让我们来实现一个简单的字符串数据类型，这个数据类型将在接下来的集合中用到。对于新的字符串，我们分配一个动态缓存来保存字文本。当这个字符串被删除时，我们将回收其所占用的内存缓冲。

`new()` 负责创建一个对象，`delete()` 必须回收这个对象所占用的资源。`new()` 预先知道它所创建的资源的类型，因为它的第一个参数将传递对这个对象的描述。基于这个参数，我们可以使用一系列的判断语句 `if` 来处理单个不同的要创建的对象。这样做的缺点是 `new()` 得完全包含对每个支持的对象要处理的代码。

现在 `new()` 拥有一个更大的问题。它负责创建对象并且返回对象的指针，这个指针被传递到 `delete()` ，也就是说，`new()` 必须在每个对象中安装特定的析构器信息。最明显的应用是使用一个指针，指向特定的析构器，这个析构器是类型描述符的一部分，被传进 `new()` 。到目前为止，我们需要像如下的声明:

```
struct type{
    size_t size;            /*size of an object*/
    void (*dtor)(void*);    /*destructor*/
};
struct String{
    char *text;             /*dynamic string*/
    const void* destroy;    /*locate destructor*/
};
struct Set{
    ...information...
    const void * destroy; /*locate destructor*/
};
```

似乎我们又有另外一个问题：在新的对象中，有人需要把析构器指针 `dtor` 从类型描述中拷贝到 `destroy` 中。而且这样的拷贝在不同的对象的类中会放到不同的位置。

初始化工作是 `new()` 的一部分，不同的类型会要求 `new()` 干不同的工作—— `new()` 可能需要不同的参数来处理不同的类型：

```
new(Set);              /*make a set*/
new(String,"text"); /*make a string*/
```

对于初始化我们使用另外一个指定类型的功能函数，这个函数我们称它为构造器。因为构造器和析构器都是类型指定的，不需要改变，我们把他们当成类型描述的一部分传递给 `new()` 函数。

注意，对于一个对象自身来说构造器和析构器并不是用来担当获取和释放内存的责任——这是 `new()` 和 `delete()` 的工作。构造器被 `new()` 调用仅仅用来初始化 `new()` 所分配的内存。对于一个字符串，这需要涉及需要另外一块内存存储文本，但是对于 `struct String` 自身的内存却是使用 `new()` 来分配的。这个空间接下来会被 `delete(`) 所释放。然而 `delete()` 首先 会调用析构器，析构器是对构造器所做的初始化进行反向操作。这步完成后才调用 `delete()` 释放 `new()` 所分配的内存。

2.2 方法，消息，类和对象

`delete()` 必须能够在不知道对象类型的情况下定位析构器。因此修正了2.1 部分的声明，我们必须坚持指针被用来定位析构器，这个指针必须放到所有对象的开始处传进 `delete()` 中，而不管他们的类型是什么。

这个指针应该指向什么呢 ？如果所有我们所拥有的是一个对象的地址，那么对于一个对象来说，指针给了我们访问指定类型的对象信息，就像对象的析构函数一样。似乎很可能我们将要创造一个类型指定的功能如一个函数用来显示对象，或者一个对象的比较函数 `differ()` ，或一个函数 `clone()` 去创建一个对象的完全拷贝。因此我们将使用一个指针指向函数指针表。

仔细看，我们会意识到这个表必须是类型描述的一部分，被传进 `new()` , 显然的解决问题的答案就是让对象指向一个类型的完全描述：

```
struct Class {
       size_t size;
       void * (* ctor) (void * self, va_list * app);
       void * (* dtor) (void * self);
       void * (* clone) (const void * self);
       int (* differ) (const void * self, const void * b);
};
struct String {
       const void * class;  /* must be first */
       char * text;
};
struct Set{
    const void* class   /*must be first*/
        ...
};
```

每个对象将以一个指针开始，这个指针向对象的类型描述表，通过这个类型描述表，我们就可以定位一个对象的类型指定信息： `.size` 是 `new()` 所分配的对象所占用内存的大小； `.dtor` 指向被 `delete()` （ `delete` 用来销毁对象） 所调用的析构器；而 `.differ` 指向一个函数，这个函数用于比较对象。

继续往下看，我们会注意到，每个功能都是以对象而存在的，通过对象来选择这些功能。只有构造函数要处理部分初始化内存区域工作。我们都叫这些功能为一个对象的方法。调用一个对象的方法就是处理一则消息，我们使用参数 `self` 来标记消息接收的对象。因为我们使用基本的C函数功能，`self` 是不需要作为第一个参数而传进的。

多个对象共享相同的类型描述符，也就是说，他们需要相同数量大小的内存空间，可用于相同的方法。我们称所有拥有相同的类型描述符的对象为一类；单独的对象被称为类的实例。到目前为止，一个类，一个抽象数据类型，可能的值与操作结合的集合，即，一个数据类型，这些是极其相似的。

一个对象是一个类的实例，也就是说，它拥有一个描述，这个描述被 `new()` 所分配的内存所指示，并且这个描述被类的方法操作。普遍来说，一个对象是特殊数据类型的值。

2.3 选择器，动态链接，多态

谁来邮递消息呢？构造器被 `new()` 所调用，对于大多数内存区域是不被初始化的:

```
void * new (const void * _class, ...)
{
    const struct Class * class = _class;
       void * p = calloc(1, class -> size);

       assert(p);
       * (const struct Class **) p = class;

       if (class -> ctor)
       {
        va_list ap;

              va_start(ap, _class);
              p = class -> ctor(p, & ap);
              va_end(ap);
       }
       return p;
}
```

在一个对象的起始地方，`struct Class` 指针的存在是极其重要的。这也是我们在 `new()` 中初始化它的原因：

如上图右边的类型描述 `class` 在编译的时候已经被初始化。对象是在运行时被创建的，接下来图中的虚线关联才被插入。在语句：

```
* (const struct Class **) p = class;
```

中，`p` 指向对象的内存区域的起始位置。我们对 `p` 进行了强制类型转换，`p` 把对象的起始位置当成一个指针，指向 `struct Class`，即把参数 `class` 设置为这个指针的值。

接下来，若构造器是类型描述的一部分，我们调用它，并把其返回值做为 `new()` 的结果，即作为一个新的对象返回。2.6 部分列出一个很聪明的构造器，由于它聪明，所以能够对它自己的内存管理作出决策。

注意啦，只有明确的可见函数如 new() 能拥有可变的参数列表。参数列表被 `va_list` 的变量 `ap` 所访问，`ap` 被一个宏 `va_start()` 初始化，这个宏在`stdarg.h` 头文件。`new()` 仅仅能够把整个参数列表传进构造器中；因此，`.ctor` 也被声明成拥有 `va_list` 的参数，而不是它私有的参数列表。由于我们接下来要在好多函数中共享源参数列表，因此我们只传递 `ap` 的地址到构造器中——当它返回时，`ap` 指向参数列表的第一个参数，而参数列表本身不会被改变。

`delete()` 假设每个对象，也就是说，每个非空指针，指向一个类型描述。如果类型描述的析构器存在，则调用它。这里，`self` 扮演前面 `p` 的角色。我们使用局部变量 `cp` 来进行强制类型转换，并从`self` 中获得我们所需要的信息。

```
void delete (void * self)
{
    const struct Class ** cp = self;

       if (self && * cp && (* cp) -> dtor){
              self = (* cp) -> dtor(self);
       }
       free(self);
}
```

析构器，在上述`delete()`中，也会获得一次把他的返回值传进`free()` 的机会，如果构造器试着去欺骗，则析构器会有更改的机会，参看2.6部分。如果一个对象在调用`delete()` 的时候不想被删除，则可在他的析构器中返回一个空指针。

所有其他的方法都存储在类型描述中，并以相似的方式被调用。在每个例子中，我们有一个单独的接收对象`self` 且我们通过它来路由我们的方法调用。

```
int differ (const void * self, const void * b)
{
    const struct Class * const * cp = self;

       assert(self && * cp && (* cp) -> differ);
       return (* cp) -> differ(self, b);
}
```

最关键的部分，当然是一个假设，假设我们能够找到一个类型描述指针 `*self` ，而这个 `*self` 会隐藏在任意的指针`self`下面。此时此刻，至少，我们会对空指针很警惕。在每个类型描述的起始，我们将存放一个“魔法数字”，或甚至把地址或所有已知类型的地址范围与 `*self` 相比较，但是，在第八章会看到，我们将做更严格的检查。

不管怎么说，`differ()` 列举出了函数调用技术怎么被动态链接或后期链接调用的原因：即只要我们能够在一开始拥有一个正确的类型描述指针，那么我们就可以对任意的对象使用`differ()` 调用。这个函数实际上被调用的时机是尽可能的晚的——即仅仅在实际执行期间

调用，而不是之前调用。

我们可以称 `differ()` 为一个选择器。它是多态功能的一个例子，也就是说，一个函数能够接受不同的参数类型，且表现不同，并且这种现象是基于他们的参数类型。一旦我们实现了更多的类时，这些类在他们的描述符中都包含 `.differ` ，则可称 `differ()` 为一个泛函数，且在这些类中能够被应用于任何对象。

我们可以把这个选择器当成方法，方法自己本身不会动态链接，但仍然能够像多态函数一样的表现，因为它能让动态的连接的函数做他们真实的事情。

多态机制实际已经嵌入到很多编程语言中，例如：如在Pascal（一种编程语言）中，`write()` 函数会根据参数类型不同进行不同的处理。在C++中，操作符 + 如果被不同的类型值如整型，指针，浮点指针调用，将产生不同的结果。这个现象被称作重载，即：参数类型和操作符名结合起来决定操作结果。相同的操作符与不同的参数类型结合将产生不同的响应。

这里并没有明显的差异。因为动态连接，`differ()` 的表现更像一个重载函数，而且C的编译器也能够使得 + 看起来像多态函数——至少对于内嵌的数据类型来说。然而，C编译器能够根据对 + 操作符的不同使用而产生不同的返回类型，但是函数 `differ()` 依靠它的参数类型只能返回相同的类型。

很多方法在不需要动态连接的情况下能够实现多态。例如，函数 `sizeOf()` 返回任意类型的对象的大小。

```
size_t sizeOf (const void * self)
{
    const struct Class * const * cp = self;

       assert(self && * cp);
       return (* cp) -> size;
}
```

所有的对象都携带它们的描述符，我们可以使用描述符来获得对象的大小。注意如下的不同之处：

```
void* s=new(string, "text");
assert(sizeof s!=sizeOf(s));
```

`sizeof` 是C语言的操作符，用于在运行时以字节的个数返回参数的大小。而 `sizeOf()` 是我们实现的多态函数，它的参数指向一个对象，返回在运行时对象所占用的字节大小。

2.4 应用

然而我们还没有实现一个字符串类，我们仍然做好了一个简单的测试程序的准备。`String.h` 定义了抽象数据类型：

```
extern const void * String;
```

对于所有的对象，我们的方法都是相似的。我们向内存管理头文件 `new.h` 中增加在1.4 部分介绍的声明：

```
void * (* clone) (const void * self);
int (* differ) (const void * self, const void * b);

size_t sizeOf(const void* self);
```

前两个源型声明称为选择器，它们在相关的 `struct Class` 中声明。下面是其应用：

```
int main ()
{
    void * a = new(String, "a"), * aa = clone(a);
       void * b = new(String, "b");

       printf("sizeOf(a) == %lu/n", (unsigned long)sizeOf(a));
       if (differ(a, b)){
              puts("ok");
       }
       if (differ(a, aa)){
              puts("differ?");
       }
       if (a == aa){
              puts("clone?");
       }
       delete(a), delete(aa), delete(b);
       return 0;
}
```

我们创建了两个字符串，并且拷贝了其中一份。我们打印出 `String` 对象所占用的大小——并不是对象的控制文本所占用的大小。最终，检查拷贝的对象与对象本身相等，但并不相同，最后再次删除字符串对象。如果所有的程序均以实现，程序的运行结果如下：

```
sizeOf(a)==8
ok
```

2.5 实现—— `String`

我们通过写这些方法实现字符串，这些方法需要被放入类型描述 `String` 中。对于实现一个新的数据类型，动态连接使我们清晰的确定出那些功能函数需要实现。

构造器从新获得文本，传递给 `new()` ，并把这些动态拷贝存储进通过 `new()` 创建的 `struct String` 中。

```
struct String {
        const void * class;  /* must be first */
        char * text;
};

static void * String_ctor (void * _self, va_list * app)
{      struct String * self = _self;
        const char * text = va_arg(* app, const char *);

        self -> text = malloc(strlen(text) + 1);
        assert(self -> text);
        strcpy(self -> text, text);
        return self;
}
```

在构造器中，我们紧紧需要初始化 .text 因为 new() 已经建立了 .class 。

析构器释放被字符串控制的动态内存。由于 delete() 只在 self 为非空的情况下调用 析构器，所以我们不需要做其他参数检查，代码如下：

```
static void * String_dtor (void * _self)
{      struct String * self = _self;

        free(self -> text), self -> text = 0;
        return self;
}
```

`String_clone()` 是对字符串的一个拷贝。接下来，源和源的拷贝都将被传进 `delete()` 中，因此我们必须对字符串的文本做一个动态内存的拷贝。这个工作通过调用 `new()` 很容易实现。

```
static void * String_clone (const void * _self)
{      const struct String * self = _self;

        return new(String, self -> text);
}
```

毫无疑问，对于 `String_differ` 如果我们比较同一个字符串对象，则返回假，若果我们比较两个不同的字符串对象，返回真，如果我们想比较字符串文本的差异可试着使用 `strcmp()`：

```
static int String_differ (const void * _self, const void * _b)
{      const struct String * self = _self;
        const struct String * b = _b;

        if (self == b)
                return 0;
        if (! b || b -> class != String)
                return 1;
        return strcmp(self -> text, b -> text);
}
```

类型描述符是独一无二的——这里我们要确定一个因素，即：我们的第二个参数是否为字符串文本。

所有这些方法都应该使用关键字 `static` 来修饰。因为这些方法只能通过 `new()` 和 `delete()` ，或者选择器调用。对于通过类型描述符的方式指定的选择器都是可用的方法。

```
#include "new.r"
static const struct Class _String = {
        sizeof(struct String),
        String_ctor, String_dtor,
        String_clone, String_differ
};

const void * String = & _String;
```

在String.h 中声明String.c 中包含的公有方法，new.h 中声明new.c中包含的公有方法。以便于正确的初始化类型描述符，这里也包含了一个私有的头文件 new.r ，此文件中包含了2.2 部分定义的 struct Class 类型描述。

2.6 另一种实现——原子

为了列举我们通过构造器和析构器到底能够做什么，我们实现了原子，所谓原子就是一个唯一的字符窜对象；如果两个原子包含相同的字符窜，则他们是相等的。原子是很容易比较的：如果两个参数的指针不同，则 `differ()` 返回真。原子的构造和销毁要付出一定的代价；我们为所有的原子维持了一个循环链表，并计数原子被克隆的次数，如下：

```
struct String {
        const void * class;                /* must be first */
        char * text;
        struct String * next;
        unsigned count;
};

static struct String * ring;       /* of all strings */

static void * String_clone (const void * _self)
{       struct String * self = (void *) _self;

        ++ self -> count;
        return self;
}
```

所有的原子的循环链表被 `ring` 所标记，通过它的成员 `.next` 来扩展，并使用构造器和析构器来维持。在构造器保存文本之前，首先会遍历链表是否有相同的文本已经存在，如下的代码插入到 `String_ctor()` 之前：

```
if (ring)
{
struct String * p = ring;
        do{
                if (strcmp(p -> text, text) == 0)
                {
++ p -> count;
                        free(self);
                        return p;
                }
        }while ((p = p -> next) != ring);
}
else{
        ring = self;
}
self -> next = ring -> next, ring -> next = self;
self -> count = 1;
```

如果我们找到了相同文本的原子，则增加它的引用计数 `count` 值，释放新的对象 `self` 返回当前找的的原子指针 `p` 。否则我们向循环链表中插入一个新字符串对象并设置其引用计数为 `count` 为1。

析构器防止删除引用计数为非零的原子。如下的代码被插入到 `String_dtor()` 之前：

```
if (-- self -> count > 0){
        return 0;
}
assert(ring);
if (ring == self){
        ring = self -> next;
}
if (ring == self){
        ring = 0;
}
else{
struct String * p = ring;
        while (p -> next != self){
p = p -> next;
                assert(p != ring);
        }
        p -> next = self -> next;
}
```

如果对引用计数的减1操作计数扔为正数，则返回一个空指针，以便于 `delete()` 手下留情。否则如果我们的字符串对象是最后一个对象我们清除循环链表标记符，否则从链表中删除我们的字符串。

把上述的实现加入到 2.4 的程序中，注意，对一个字符串对象的克隆此时为源字符串对象本身，运行结果如下：

```
sizeOf(a)==16
ok
clone?
```

2.7 总结

给一个指针指定一个对象，动态连接使我们找到了类型指定的函数功能：每个对象都会以一个描述符开始，这个描述符包含了指针，指向对象的可用函数指针表。尤其是，一个描述符包含一个指向构造器的指针，这个构造器用来初始化对象所关联的内存区域，另外这个指针还指向一个析构器，析构器会在删除对象之前回收对象所拥有的资源。

我们称所有的对象所共享的描述符为一个类。而对象是类的实例，对于对象指定类型的功能被称作对象的方法，而消息被这些功能函数所调用。对于一个对象，我们使用选择器功能去定位和调用动态连接的方法。

通过选择器和动态连接使得相同函数名对于不同的类而产生不同的结果。这样的函数被称为是多态的。

多态功能是非常有用的。他们提供了一种概念上的抽象：`differ()` 可比较任何两个对象——我们不需要铭记`differ()` 针对具体的情形是否可用。一个很容易并非常方便的调试工具就是多态函数 `store()` ，可在一个文件描述符上显示任何对象。

2.8 练习

了解了多态的功能后，我们需要使用动态连接来实现`Object` 和 `Set` 。这对于 `Set` 来说是比较困难的。我们不再记录一个元素属于哪个集合。

对于字符串来说，似乎有更多的方法去实现。我们需要知道字符串的长度，我们更想为一个对象从新设置它的字符串文本，我们应该能打印字符串文本。如果我们乐意去处理子串，将会更加有趣味。

原子是如此有效的，我们可使用一个哈希表来跟踪它，那么一个原子的值能否被改变呢？

`String_clone()` 呈现出一个微妙的问题：在这个函数中，`String` 的值似乎应该与`self->class` 相同。我们向 `new()` 中传递的参数会有任何变化吗？

第三章 编程的悟性——算术表达式

> 来源：http://blog.csdn.net/besidemyself/article/details/6423491
>
> 译者：besidemyself

动态连接就其本身而言是一项强大的编程技术，并不是去写一些带有庞大的 `switch` 语句去处理很多特例的函数。我们可以写很多小的函数，对于每个 `case` 语句，安排适当的函数被动态连接调用。这样做通常简化了编程工作并且会使得代码容易扩展。

作为一个例子，我们将写一个小的程序去读并评估由浮点数字，括号，常用操作符，减号，等组成的算术表达式。正常情况下我们宁愿使用编译器产生器工具 `lex` 和 `yacc` 去建立这部分的程序去剖析算术表达式。这不是一本关于编译器建立的书，然而，就仅仅此次我们将自己写这次的代码。

3.1 主循环

程序的主循环从标准输入读取一行数据，初始化以便数字和操作符能被提取出来，空格被忽略，调用一个函数去确认正确的算术表达式并存储之，最终处理所存储的表达式。如果出错了，我们简单的读取下一行数据。如下为主循环：

```
#include <setjmp.h>
int main (void)
{
    volatile int errors = 0;
       char buf [BUFSIZ];

       if (setjmp(onError)){
              ++ errors;
       }

       while (fgets(buf, sizeof buf, stdin)){
              if (scan(buf))
              {     void * e = sum();

                     if (token){
                            error("trash after sum");
                     }
                     process(e);
                     delete(e);
              }
}
       return errors > 0;
}
void error (const char * fmt, ...)
{
    va_list ap;

       va_start(ap, fmt);
       vfprintf(stderr, fmt, ap), putc('/n', stderr);
       va_end(ap);
       longjmp(onError, 1);
}
```

错误恢复点被使用 `setjmp()` 所定义。如果 `error()` 在程序中的某个位置被调用，`longjmp()` 伴随着从 `setjmp()` 另外一个返回而继续执行。在这种情况下，结果是一个值被传进 `longjmp()` ，错误累加，而且下一个输入行被读取。如果遇到错误，程序的出口代码将报告错误。

3.2 扫描器

在主循环中，一旦一个输入行被读入到 `buf[]` 中，它将被传进 `scan()` , 此函数对于每一个调用把下一个输入符号放入变量 `token` 。在最后一行， `token` 的值为0：

```
#include <ctype.h>
#include <errno.h>
#include <stdlib.h>
#include “parse.h”

static double number;           /* if NUMBER: numerical value */
static enum tokens scan (const char * buf)
{
    static const char * bp;

       if (buf){
              bp = buf;               /* new input line */
       }
       while (isspace(* bp & 0xff)){
              ++ bp;
       }
       if (isdigit(* bp & 0xff) || * bp == '.')
       {
        errno = 0;
              token = NUMBER, number = strtod(bp, (char **) & bp);
              if (errno == ERANGE){
                     error("bad value: %s", strerror(errno));
              }
       }
       else{
              token = * bp ? * bp ++ : 0;
       }
       return token;
}
```

我们调用 `scan()` ，可传递输入行缓冲的地址，或传进一个空指针得以继续工作在当前的行。空格被忽略，并且遇到第一个为数字或小数点，我们就是用一个ANSI-C 的函数 `strtod()` 开始提取出浮点数字。若为其他的任何字符将被返回，并且我们不会预先在输入缓冲传递一个空字节。

`scan()` 的结果被存储在全局变量 `token` ——这样简化了识别程序（识别器）。如果我们侦测出一个数字，我们将返回唯一的值 `NUMBER` 并使得在全局变量 `number` 中实际的值有效。

3.3 识别器

在最高水平,表达式通过函数 `sum()` 被识别， `sum()` 函数内部调用 `scan()` 并返回一个表示，这个表示可通过调用 `process()` 被处理并通过 `delete()` 被回收。

如果我们不使用 `yacc` (是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器））,我们将通过递归下降的方法识别表达式，合乎语义的规则被翻译成等价的C函数。例如：一个 `sum` 是一个产物，接下来被0跟随，或更多的组，每个由额外的操作符和另外的产物组成，一个语义规则如下：

```
sum：product {+|- product}…
```

被翻译成C函数如下：

```
static void * sum (void)
{
    void * result = product();
       const void * type;

       for (;;)
       {     switch (token) {
              case '+':
              case '-':
                     scan(0),product();continue;
              return;
       }
}
```

对于每一个语义规则有一个C函数，以便于这些规则能够相互调用，这些不同的分支被转换成 `switch` 或 `if` 语句，迭代的语法将在C中翻译成循环。仅仅一个问题就是我们必须避免无限的递归。

`token` 总是包含下一个输入的符号。如果我们识别出它，我们必须调用 `scan(0)`

3.4 处理器

我们如何来处理表达式呢？如果我们仅仅想用一些用数字表示的值执行简单的算术。我们可以扩展识别函数并且一旦识别出操作符和操作码就计算出结果如： `sum()` 应该会期望从每一个对 `product()` 的调用期望一个 `double` 类型的结果，尽可能的执行加或减法，并且返回结果，再次作为一个 `double` 类型函数的值。

如果我们想要建立一个系统用来处理更加复杂的表达式，我们需要存储表达式以便于后续处理。在这种情况下，我们能够不仅仅执行算术，而且可以允许决定并且有条件的评估一个表达式的一部分，且可用存储的表达式作为用户的函数包含在其他表达式中。我们所需要的是一个合理通用的方式代表一个表达式。比较常规的技术是使用一个二叉树在每一个节点上存储 `token` .

```
struct Node {
    enum tokens token;
    struct Node * left, * right;
};
```

然而，这样并不是很灵活。我们需要介绍一个 `union` 去创建一个节点，在这个节点上我们可存储一个数，并且我们在这些节点代表的一元操作符上浪费了空间。此外，`process()` 和 `delete()` 将包含 `witch` 分支，并 `witch` 分支会随着我们增加的符号而增多。

3.5 信息隐藏

应用迄今为止我们学到的，我们绝不去揭示节点结构。相反，我们先在头文件 `value.h` 中放置一些声明如下：

```
const void * Add;
    …
void * new (const void * type, ...);
void process (const void * tree);
void delete (void * tree);
```

现在我们可以编写代码 `sum()` 如下：

```
#include "value.h"
static void * sum (void)
{
    void * result = product();
       const void * type;

       for (;;)
       {
        switch (token) {
                  case '+':
                         type = Add;
                         break;
                  case '-':
                         type = Sub;
                         break;
                  default:
                         return result;
              }
              scan(0);
              result = new(type, result, product());
       }
}
```

`product()` 与 `sum()` 有相同的结构，并且调用 一个函数 `factor()` 去识别数字，符号，且 `sum` 被赋予了括号：

```
static void * factor (void)
{
    void * result;

        switch (token) {
        case '+':
                scan(0);
                return factor();
        case '-':
                scan(0);
                return new(Minus, factor());
        default:
                error("bad factor: '%c' 0x%x", token, token);
        case NUMBER:
                result = new(Value, number);
                break;
        case '(':
                scan(0);
                result = sum();
                if (token != ')')
                        error("expecting )");
        }
        scan(0);
        return result;
}
```

尤其在 `factor()` 中，我们需要特别小心的保持扫描器（scanner）是不变的：`token` 必须总是包含下一个输入的符号。一旦 `token` 被使用，我们需要调用 `scan(0)` 。

3.6 动态连接

识别器是完善的。`value.h` 对于算术表达式完全隐藏了求值程序，且与此同时指定了我们必须所实现的。 `new()` 携带描述符，如 `Add` 和合适的参数如指针对加的操作且返回一个表示和的指针。

```
struct Type {
        void * (* new) (va_list ap);
        double (* exec) (const void * tree);
        void (* delete) (void * tree);
};

void * new (const void * type, ...)
{
va_list ap;
        void * result;

        assert(type && ((struct Type *) type) -> new);

        va_start(ap, type);
        result = ((struct Type *) type) -> new(ap);
        * (const struct Type **) result = type;
        va_end(ap);
        return result;
}
```

我们使用动态连接并传递一个对指定节点例程的调用，在例程中的 `Add` 分支处，必须常见一个节点，并且传进两个指针。

```
truct Bin {
        const void * type;
        void * left, * right;
};

static void * mkBin (va_list ap)
{
struct Bin * node = malloc(sizeof(struct Bin));

        assert(node);
        node -> left = va_arg(ap, void *);
        node -> right = va_arg(ap, void *);
        return node;
}
```

注意，只有 `mkBin()` 知道它创建的是什么。所有我们要求的是各个节点对于动态连接是以一个指针开始。这个指针被 `new()` 传进一遍于 `delete()` 能够调用到它指定节点的函数：

```
void delete (void * tree)
{
   assert(tree && * (struct Type **) tree
           && (* (struct Type **) tree) -> delete);

   (* (struct Type **) tree) -> delete(tree);
}
```

动态连接很优雅的避免了复杂难解的节点。`.new()` 精确的创建了每个类型描述符的右节点：二元操作符拥有两个子孙。一元操作符拥有一个子孙，且值节点仅仅包含了值。`delete()` 是一个非常简单的函数因为每个节点处理它自己的销毁过程：二元操作符删除两个子树并且释放他们自己的节点，一元操作符仅仅删除一个子树，且值节点仅仅释放自己。变量和常量甚至可以留到后面——对于 `delete()` 的回应他们简单的什么也不做。

3.7 A Postfix Writer

到目前为止我们还没有真正的决定 `process()` 将要真正做什么。如果我们想要发布一个表达式的后缀版，我们将要对 `struct Type` 增加一个字符串以便于显示出实际的操作符，且 `process()` 将要安排一个单独的被 `tab` 键缩进的行：

```
void process (const void * tree)
{
        putchar('/t');
        exec(tree, (* (struct Type **) tree) -> rank, 0);
        putchar('/n');
}
```

`exec()` 处理动态连接

```
static void exec (const void * tree, int rank, int par)
{
        assert(tree && * (struct Type **) tree
                && (* (struct Type **) tree) -> exec);

        (* (struct Type **) tree) -> exec(tree, rank, par);
}
```

每一个二元操作符被使用如下函数发出：

```
static void doBin(const void *tree)
{
    exec(((struct Bin *) tree) —> left);
    exec(((struct Bin *) tree) —> right);
    printf(" %s", (* (struct Type **) tree) —> name);
}
```

类型描述符如下绑定：

```
static struct Type _Add = { "+", mkBin, doBin, freeBin };
static struct Type _Sub = { "—", mkBin, doBin, freeBin };
const void * Add = & _Add;
const void * Sub = & _Sub;
```

应该很容易猜测一个数值是怎样被实现的。它被代表作为一个结构体携带 `double` 类型的信：

```
struct Val {
    const void * type;
    double value;
};
static void * mkVal (va_list ap)
{
    struct Val * node = malloc(sizeof(struct Val));
    assert(node);
    node —> value = va_arg(ap, double);
    return node;
}
```

处理组成的打印值：

```
static void doVal (const void * tree)
{
    printf(" %g", ((struct Val *) tree) —> value);
}
```

我们已经做了——没有子树要删除，因此我们可以使用库函数 `free()` 直接的删除值节点：

```
static struct Type _Value = { "", mkVal, doVal, free };
const void * Value = & _Value;
```

一元操作符如 `Minus` 将留作练习。

3.8 算术

如果我们想做算术运算，我们让执行的函数返回一个 `double` 类型的值，然后让 `process()` 打印这个值：

```
static double exec (const void * tree)
{
    return (* (struct Type **) tree) —> exec(tree);
}
void process (const void * tree)
{
    printf("/t%g/n", exec(tree));
}
```

对于每个节点的类型，我们需要一个执行函数来计算和返回这个节点的值。这里有两个实例：

```
static double doVal (const void * tree)
{
    return ((struct Val *) tree) —> value;
}
static double doAdd (const void * tree)
{
    return exec(((struct Bin *) tree) —> left) +
    exec(((struct Bin *) tree) —> right);
}
static struct Type _Add = { mkBin, doAdd, freeBin };
static struct Type _Value = { mkVal, doVal, free };
const void * Add = & _Add;
const void * Value = & _Value;
```

3.9 插入输出

也许对于处理算术表达式的突出点是带小括号的形式打印。这通常是有点滑稽的，依照谁来负责发出括号。此外对于操作符的名字用于前缀输出，我们增加了两个数值到 `struct Type` 中。

```
struct Type {
    const char * name; /* node’s name */
    char rank, rpar;
    void * (* new) (va_list ap);
    void (* exec) (const void * tree, int rank, int par);
    void (* delete) (void * tree);
};
```

`.rank` 是优先的操作符，以1开始，此外 `.rpar` 被设置用于操作符，如减操作，此操作如果用于相等的优先级的操作就要求他们的右操作被附上括号。

```
$ infix
1 + (2 – 3)
1 + 2 – 3
1 – (2 – 3)
1 – (2 – 3)
```

这个证实了我们需要如下的初始化：

```
static struct Type _Add = {"+", 1, 0, mkBin, doBin, freeBin};
static struct Type _Sub = {"–", 1, 1, mkBin, doBin, freeBin};
```

滑稽的部分是对于二元节点得去决定它是否必须要增加括号。一个二元节点如加法，被给予它自己较高的优先级并且一个标记指示在相等的优先级中括号是否是必须的。 `doBin()` 去判别是否使用括号:

```
static void doBin (const void * tree, int rank, int par)
{
    const struct Type * type = * (struct Type **) tree;
    par = type –> rank < rank
    || (par && type –> rank == rank);
    if (par)
    putchar('(');
    exec(((struct Bin *) tree) –> left, type –> rank, 0);
    printf(" %s ", type –> name);
    exec(((struct Bin *) tree) –> right,
    type –> rank, type –> rpar);
    if (par)
    putchar(')');
}
```

与高优先级的操作符比若我们有一个较低优先级，或者如果我们被要求在相等的优先级情况下输出括号，我们就打印括号。在任何情况下，如果我们的描述有 `.rpar` 的设置，我们要求仅仅我们的所有操作输出额外的括号如上：

保持打印的实例程序是较容易写的。

3.10 总结

三种不同的处理器证实了信息隐藏的优越性。动态连接帮助我们把一个问题分解成很简单的函数功能点。最终的程序是很容易扩展的——试着去增加C语言中的比较和如 `?:` 的操作符吧。

第四章 继承——代码重用和改进

> 来源：http://blog.csdn.net/besidemyself/article/details/6423491
>
> 译者：besidemyself

4.1 一个超级类——点

我们将在这章以一个基本的画图程序作为开始。这里是是我们乐意拥有的其中一个类的快速测试如下：

```
#include "Point.h"
#include "new.h"

int main (int argc, char ** argv)
{
    void * p;

       while (* ++ argv)
       {
        switch (** argv) {
                  case 'p':
                         p = new(Point, 1, 2);
                         break;
                  default:
                         continue;
              }
              draw(p);
              move(p, 10, 20);
              draw(p);
              delete(p);
       }
       return 0;
}
```

对于每一个命令参数以字符 `p` 开始，我们获得一个新的绘图的点，移动这个点到某处，从新绘制，并且删除。标准化C语言不包含图形化输出标准的函数：然而，如果我们坚持产生一幅图片，我们能够发表文本，对于这个文本Kernighan 的图片 [Ker82] 能够理解：

```
$ points p
"." at 1,2
"." at 11,22
```

坐标对于测试是无关紧要的——从商业和面向对象的说法解释："点就是一则消息。"

我们用这个点能做些什么呢？ `new()` 将产生一个点，并且构造器期望着初始化坐标作为进一步的参数传进 `new()` 。通常， `delete()` 将回收我们的点并且按照惯例调用析构器。

`draw()` 安排点被显示出来。由于我们希望与其他图形对象协同工作——因此在测试程序中会有 `switch` ——对于 `draw()` 我们将提供动态连接。

`move()` 通过传递一系列参数来改变点的坐标。如果我们实现每一个图形对象，这些对象都与它涉及的点关联，我们将能够通过简单的应用这个点的 `move()` 方法来移动它。因此，对于 `move()` 在不需要动态连接的情况下我们应该可以做。

4.2 超级类的实现——点

在 `Point.h` 中，抽象数据类型包含如下：

```
extern const void * Point;                    /* new(Point, x, y); */

void move (void * point, int dx, int dy);
```

我们能够重复利用第二章的 `new` 文件，尽管我们删除了很多方法并且对 `new.h` 文件增加了 `draw()` 方法：

```
void * new (const void * class, ...);
void delete (void * item);
void draw (const void * self);
```

在 `new.r` 中类型描述 `struct Class` 应该与在 `new.h` 中声明的方法相关联：

```
struct Class {
        size_t size;
        void * (* ctor) (void * self, va_list * app);
        void * (* dtor) (void * self);
        void (* draw) (const void * self);
};
```

选择器 `draw()` 在 `new.c` 中实现。它将代替如 `differ()` 在2.3 节介绍的选择器，并且以相同的风格编写代码：

```
void draw (const void * self)
{
    const struct Class * const * cp = self;

        assert(self && * cp && (* cp) -> draw);
        (* cp) -> draw(self);
}
```

这些预备工作完成后，我们将转去做真正的工作去写 `Point.c` ，对点的实现。在此，面向对象帮助我们精确的鉴别出我们需要做什么：我们必须对表示式做出决定并实现构造器，析构器，动态链接方法 `draw()` 和静态链接方法 `move()` ，这些都是基本的函数。如果我们坚持二维，笛卡尔坐标，我们选择如下明确的表示：

```
struct Point {
        const void * class;
        int x, y;                       /* coordinates */
};
```

构造器必须初始化坐标 `.x` 和 `.y` ——现在一个绝对的例程如下：

```
static void * Point_ctor (void * _self, va_list * app)
{
    struct Point * self = _self;

       self -> x = va_arg(* app, int);
       self -> y = va_arg(* app, int);
       return self;
}
```

现在的结果是我们并不需要析构器，因为在 `delete()` 之前没有资源需要回收。
在 `Point_draw()` 函数中，我们以一种图片能够识别的方式打印当前的坐标：

```
static void Point_draw (const void * _self)
{
    const struct Point * self = _self;

       printf("\".\" at %d,%d\n", self -> x, self -> y);
}
```

这样照顾到所有的动态连接方法，并且我们能够定义类型描述符，在此一个空的指针代表一个不存在的析构器：

```
static const struct Class _Point = {
       sizeof(struct Point), Point_ctor, 0, Point_draw
};

const void * Point = & _Point;
```

`move()` 不是动态连接的，因此我们省略 `static` 使得它作用域能够超出 `Point.c` 并且我们不给它加类名前缀 `Point` ：

```
void move (void * _self, int dx, int dy)
{
    struct Point * self = _self;

       self -> x += dx, self -> y += dy;
}
```

与在 `new.c` 中的动态连接相结合，这就得出了 `Point.c` 中点的实现。

4.3 继承——环

一个环形仅仅是一个大的点：此外对于中心坐标它需要一个半径。画法有点不同，但是移动只需要我们改变中心坐标。

这就是我们能够正常的为我们的文本编辑器和演择源代码重用而做好准备的地方。我们对点的实现做一个拷贝并且改变环与点不同的地方。 `Struct Circle` 获取其他额外的组成：

```
int rad;
```

这部分组成在构造器中初始化

```
self->rad=va_arg(*app,int);
```

并且在 `Circle_draw()` 中使用：

```
printf("circle at %d,%d rad %d\n",
self —> x, self —> y, self —> rad);
```

我们在 `move()` 中有点迷惑。对于一个点和一个环必要的动作是相同的：对于坐标部分我们需要增加转移参数。然而，在一种情况， `move()` 工作于 `struct Point` ，在另外一种情况，它工作与 `struct Circle` 。如果 `move()` 是动态连接的，我们需要提供两个不同的函数去做相同的事情。但是，会有更好的方式，考虑一下点和环表示的层：

```
struct Point          struct Circle
```

图片显示每一个环都以一个点开始。如果我们分配一个 `struct Circle` 通过增加到 `struct Point` 的结尾，我们可以向 `move()` 函数中传递一个环，因为表示式的初始化部分看起来仅仅像点，而 `move()` 方法期望接到收点，并且点仅仅是 `move()` 方法能够改变的。这里是一个合理的方式确保对环的初始化部分总看起来像点：

```
struct Circle { const struct Point _; int rad; };
```

我们让派生的结构体以一个我们要扩展的基结构体的拷贝而开始。信息隐藏要求我们决不直接的访问基结构体；因此，我们使用几乎不可见的下划线作为它的名字并且把它声明为 `const` 避开粗心的指派。

这就是简单的继承的全部：一个子类从一个超类（或者基类）继承仅仅通过扩充表示超类的结构体。

由于子类对象（一个环）的表示就像一个超类对象（一个点）的表示一样动身。环总能够佯装成一个点——在一个环的表示的初始化地址处的确是一个点的表示。

向 `move()` 中传递一个环是完全确定的：子类继承了超类的方法，因为这些方法仅在子类的表示上操作，这些子类的表示和超类的表示是相同的， 而这些方法原先就在超类上写好了。传递一个环就像传递一个点意味着把 `struct Circle*` 转换成 `struct Point*` 。我们将把这样的操作看成一个从子类到超类的上抛——在标准化C语言中，它能够使用明确的转换操作符来实现或者通过中间的 `void*` 的值。

这通常是不佳的，然而，传递一个点到一个函数专为环如，`Circle_draw()`：如果一个点原先就是一个环，从`struct Point*`转换成`struct Circle*`仅仅是可允许的。我们称这样的从超类到子类的转换为下抛——这也要求明确的转换或void*值，并且它仅仅对于指针，对于对象能够使用，指针，对象在子类的开始做转换。

对于动态连接方法如`draw()`，这种情形是不同的。让我们再次看先前的图片，这次完全明确类型描述符如下 ：

4.4 连接和继承

`move()`不是动态连接的并且不使用动态连接方法做工作。然而我们能够传递一个指针和环到`move()`中，它的确不是一个多肽的函数：`move()`对于不同的对象不会做不同的处理，它总是增加参数到坐标，忽略其他与坐标相依附的。

当我们上抛从一个环到一个点时，我们没有改变环的状态，换句话说，即使我们把环的`struct Circle`表示当成一个点的`struct Point`，我们不会改变它的内容。结果，把环视为点作为一个类型描述符仍然拥有`Circle`，因为点在它的`.class`部分并没有改变。`draw()`是一个选择器函数，即，它将会使用无论传入什么样的参数作为自身，去处理被`.class`所指示的类型描述符，并且调用在这里存储的画图方法。

一个子类继承它的超类的静态链接的方法——这些方法操作子类对象的部分，这些子类对象是已经在超类对象上呈现的。一个子类能够选择支持它自己的方法代替它的超类的动态连接方法。如果继承，即，若没有重写，超类动态的连接的方法就像静态连接的方法一样的起作用并且修改子类对象的超类的部分内容。如果重写，子类他自己的动态连接方法的版本访问子类对象所有的表示，即，对于一个环，`draw()`将会调用`Circle_draw()`方法，此方法能够考虑到半径当画环的时候。

4.5 静态和动态连接

一个子类继承了它的超类的静态链接的方法并且选择性的继承或重写动态连接的方法。考虑对于`move()`和`draw()`的声明如下：

```
void move(void* point, int dx,int dy);
void draw(const void* self);
```

我们不能够从这两个声明中发现连接，尽管对于`move()`的实现能够直接的工作，然而`draw()`仅仅是一个选择器函数在运行时跟踪动态连接。不同点就是我们声明一个静态链接方法就像`move()`在`Point.h`中作为抽象数据类型接口的一部分，且我们声明一个动态连接方法就像`draw()`携带内存管理接口在`new.h`中，因为迄今为止我们已经决定在`new.c`中实现数据选择器。

静态链接会更加有效率因为C编译器能够使用直接的地址调用子程序，但是对于一个函数如 `move()` 对于子类不能被重写。动态连接在间接调用的扩展上更加便捷——我们已经对调用选择器函数如 `draw()` 的额外开销作了决定，检查参数，定位，调用正确的方法。我们丢弃了检查并且使用 `macro*` 像如下减少了额外开销：

```
#define draw(self)  ((*(struct Class**)self)->draw(self));
```

但是如果他们的参数有负面的影响宏会引发问题并且对于宏并没有明确的技术用于操作可变参数列表。此外，宏需要 `struct Class` 的声明，此 `struct Class` 到目前为止对于类的实现已经可用而不是对于整个程序。

不幸的是当我们设计超类时，我们还需要决定很多事情。但是函数调用方法是不会改变的，它会占用很多文本编辑，更可能的会在许多类中，把一个函数的定义从静态转换到动态连接，反之亦然。从第七章开始我们将使用一个简单的预处理去简化编码，即使如此连接转换也是极易出错的。

带着这种怀疑，与静态链接相比决定动态连接可能会更好点即使它效率较低。通用函数能提供一个有用的概念性的抽象并且他们倾向于减少我们需要在项目过程中记忆的函数名的数量。如果，实现所有要求的类后，我们发现其实动态连接方法从来没有被重写，通过其单一的实现去替代它的选择器并且甚至在 `struct Class` 中浪费它的位置与扩展类型描述和更正所有的初始化相比麻烦会更少。

4.6 可见度和访问函数

我们现在可以尝试着实现 `Circle_draw()` 。基于“need to know ”这样的规则信息隐藏要求我们对于每个类使用3个文件。 `Circle.h` 包含抽象数据类型接口；对于一个子类它包含了超类的接口文件以便于这样的声明使得继承的方法可用：

```
#include "Point.h"
extern const void* Circle;  /*new(Circle,x,y,rad)*/
```

接口文件 `Circle.h` 被应用程序代码所包含并且对于类的实现；它避免了多次包含所引发的错误。

一个环的表示在第二个头文件中声明， `Circle.r` 。对于子类它包含了超类的表示文件以便于我们能够通过扩展超类派生出子类的表示：

```
#include "Point.r"
struct Circle{const struct Point _;int rad;};
```

子类需要超类的表示去实现继承：`struct Circle` 包含了一个 `const struct Point` 。这个点确定不是只读的—— `move()` 将改变它的坐标——但是 `const` 限定词防止了意外的覆盖它的组成部分。表示文件 `Circle.r` 仅仅被类的实现所包含；仍然受到多重调用的保护。

最终，对一个环的实现对于类，对于对象管理，被在包含接口和表示文件的原文件 `Circle.c` 中所定义：

```
#include "Circle.h"
#include "Circle.r"
#include "new.h"
#include "new.r"

static void Circle_draw(const void * _self)
{
	const struct Circle* self=_self;
	printf("circle at %d rad %d\n",self->_.x,self->_.y,self->rad);
}
```

在 `Circle_draw()` 中，对于环我们通过子类部分使用“可见的名字”. 来读取点部分。从信息隐藏的角度看这并不是一个好的注意。然而读取坐标值不应该产生重大的问题，我们决不能确保在其他情形下，一个子类的实现不去直接的欺骗和修改它的父类的一部分，因此带着其不变量去玩一场浩劫。

效率要求一个子类能直接的访问到其超类的组成部分。信息隐藏和可维护性原则要求一个超类从它的子类上尽可能好的隐藏对它自己的表示。如果我们后面做出选择，我们应该能够提供对这些子类被允许查看超类所有组成部分访问函数，并且对于这些组成部分提供更正函数，即，便要子类去做修改。

访问和修改函数时静态链接的方法。如果我们对于超类在表示文件中声明了他们，超类仅包含在子类的实现中，我们可以使用宏，如果宏使用每个参数仅以此则副作用没有问题。作为一个例子，在 `Point.r` 中，我们定义了下面的访问宏:

```
#define x(p)    (((const struct Point*)(p))->x)
#define y(p)    (((const struct Point*)(p))->y)
```

这些宏对于任何以 `struct Point` 开始对象能够被应用于一个指针，也就是说，对于对象，从我们的点的任何子类。这项技术即为，上抛我们的点到超类并引用我们感兴趣的部分。 `const` 在抛得过程中对结果的分配。如果 `const` 被忽略

```
#define x(p)    (((struct Point*)(p))->x)
```

一个宏调用 `x(p)` 产生一个能成为分配的目标的 `l-value` ，一个好点的修改函数最好是一个宏的定义

```
#define set_x(p,v) (((struct Point*)(p))->x=(v))
```

此定义产生一个分配。

在子类实现的外部对于访问和修改函数我们仅仅使用静态链接的方法。我们不能够求助于宏，因为对于宏引用超类的内部表示是不可见的。对于包含进应用程序的信息隐藏并不提供表示文件 `Point.r` 而实现。

宏定义揭示了，然而，一旦一个类的表示可用，信息隐藏能够被很容易的击败。这里有一个方式更好的隐藏 `struct Point` 。在超类的实现中，我们使用正常的定义：

```
struct Point{
        const void* class;
        int x,y;
};
```

对于子类的实现我们提供下面的看起来不透明的版本：

```
struct Point{
        const char _[sizeof(struct {const void* class; int x,y;})];
};
```

这个结构体像先前拥有相同的大小，但是我们不能够读取也不能够写它的组成部分因为他们被隐藏在一个匿名的内部结构中。重点是这两种声明必须包含相同的组成部分的声明并且这在没有与处理器的情况下是很难维持的。

4.7 子类的实现——环

我们已经做好了些完整实现的准备，我们可以选择先前部分介绍的我们最喜欢的技术。面向对象规定我们需要一个构造器，可能的话还会有一个析构器，`Circle_draw()` ，和类型描述 `Circle` 都绑定在一起。以便于练习我们的方法，我们包含了 `Circle.h` 并增加了下面的行在4.1 部分的程序中做测试：

```
case 'c':
        p=new(Circle,1,2,3);
        break;
```

现在我们能够观察到下面的测试程序的表现：

```
$ circles p c
"." at 1,2
"." at 11,12
circle at 1,2 rad 3
circle at 11,22 rad 3
```

环的构造函数接收3个参数：第一个参数为环的点的坐标接下来是半径。初始化点部分是点的构造器的工作。它会处理部分 `new()` 参数列表的参数。环的构造器从它的初始化半径的地方携带保留的参数列表。

一个子类的构造器首先应该允许超类做部分初始化，这部分初始化把清晰地内存带进超类对象。一旦超类构造器构造完成，子类构造器完成初始化并把超类对象带进子类对象中。

对于环，意味着我们需要调用 `Point_ctor()` 。像其他所有动态链接一样，这个函数被声明为 `static` ，因此隐藏在 `Point.c` 的内部。然而，我们仍然能够通过在 `Circle.c` 中可用的类型描述符来 `Point` 获得此函数。

```
static void * Circle_ctor (void * _self, va_list * app)
{
        struct Circle * self =
        ((const struct Class *) Point) —> ctor(_self, app);

        self —> rad = va_arg(* app, int);
        return self;
}
```

这里应该很清楚为什么我们传递参数的地址 `app` 列表指针到每个构造器而不是 `va_list` 的值本身： `new()` 调用子类的构造器，此构造器调用超类的构造器，等等。最超级的构造器是第一个将去实际的作一些事情，并且会捡起传进 `new()` 的最左边的参数列表。保留的参数对于下一个子类是可用的，等等知道最后，最右边的参数被最终的子类所使用，也就是说，被 `new()` 所直接的调用的构造器所调用。

构造器以严格的相反的次序是最好的组织： `delete()` 调用子类的析构器。它首先应该销毁它自己的资源接下来调用直接的超类的析构器，这个析构器可直接的销毁下一个资源集等等。构造是先发生在子类之前的父类上的。析构则是相反，子类要先于父类，即，环部分要先于点部分。这里，然而，什么也不需要做。

我们先前已经让 `Circle_draw()` 工作了，我们使用可见部分，并且编码表示文件 `Point.r` 如下：

```
struct Point {
        const void * class;
        int x, y; /* coordinates */
};
#define x(p) (((const struct Point *)(p)) -> x)
#define y(p) (((const struct Point *)(p)) -> y)
```

现在我们可以对于 `Circle_draw()` 使用访问宏:

```
static void Circle_draw (const void * _self)
{
       const struct Circle * self = _self;
        printf("circle at %d,%d rad %d\n",x(self), y(self), self —> rad);
}
```

`move()` 拥有静态链接并且被从点的实现上继承。我们得出结论环的实现是通过定义仅仅全局可见 `Circle.c` 的部分内容:

```
static const struct Class _Circle = {
	sizeof(struct Circle), Circle_ctor, 0, Circle_draw
};

const void * Circle = & _Circle;
```

然而，在接口，表示式，实现文件之间似乎我们有一个可行的分配程序文本实现类的策略，点和环的例子还没有显现出一个问题：如果一个动态连接的方法如 `Point_draw()` 在子类中没有被重写，子类的类型描述符需要指向在父类实现的函数。函数名，然而在这里被定义成 `static` ，因此选择器是不能够被规避的。我们将在第六章看到一个清晰地解决此问题的方法。作为暂时的权衡，我们在这种情况下可以避免对 `static` 的使用，仅仅在子类的实现文件中声明函数的头，对于子类并且使用函数名去初始化类型描述。

4.8 总结

超类的对象和子类是相似的，但是在表现形式上并不相同。子类正常情况下会有更详尽的陈述更多的方法——他们被超类对象的版本专用指定。

我们使用超类对象的表示的拷贝来作为子类对象表示的开始，即，子类对象通过把它的组成部分增加到超类对象的末尾被表示。

一个子类继承了超类的方法：因为一个子类对象的起始部分看起来像超类对象，我们可以上抛并且看到一个指向子类对象的指针作为一个指向我们能够传递超类方法的超类对象。为了避免显性转换，我们使用 `void*` 作为通用指针来声明所有方法的参数。

继承可以被看成一个多态机制的根本形式：一个超类方法接受不同类型，它自己的类和所有子类命名的对象。然而因为对象都佯装成超类对象，方法仅仅在每个对象的超类部分起作用，并且它将，因此从不同的类对于对象不会起不同的作用。

动态链接方法能够从一个超类继承或在子类中重写——对于子类通过无论何种函数的指针被放进类型描述符来决定。因此，对于一个对象如果动态链接方法被调用，我们总能够访问属于对象真正的类的方法即使指针上抛到一些超类上。如果动态链接方法被继承，它只能在子类对象的超类部分起作用，因为它的确不知道子类的存在。如果一个方法被重写，子类的版本能够访问整个对象，他甚至可以通过显性的超类的类型描述符的使用来调用它关联的超类的所有方法。

特别注意，对于超类的表示，构造器首先回调超类的构造器直到最终的祖先以便于每个子类的构造器仅仅处理它自己的对类的扩展。每个超类析构器应该先删除它的子类的资源然后调用超类的析构器等等直到最终的祖先。构造器的调用顺序是从祖先到最终的子类，析构器的发生则正好是相反的顺序。

我们的策略还是有点小毛病的：在通常情况下我们不应该从一个构造器中调用动态链接方法，因为对象也许并没有完全被初始化好。在构造器被调用之前 `new()` 把最终的类型描述符插入到一个对象中，作为一个构造器在相同的类中是没有必要的访问方法的。安全的技术是

在相同的类中对于构造器通过内部的名字来调用方法，也就是说，对于点，我们调用`Points_draw()`而不是`draw()`。

为了鼓励信息隐藏，我们使用了三个文件对类的实现。接口文件包含了抽象的数据类型描述，表示文件包含了对象的结构，实现文件包含了方法和初始化类型描述的代码。一个接口文件包含了超类接口文件并且被实现和任何应用所包含。一个表示文件包含了超类的表示文件并且仅仅被实现所包含。

超类的部分不应该直接的在子类中被引用。相反，对于每个部分我们能够既提供静态链接访问和尽可能的修改方法，也能对于超类的表示文件增加适当的宏。函数符号使得使用文本编辑器或调试器去跟踪可能的信息泄露或不变量的破坏更简单。

4.9 是或有吗？——继承对集合

作为`struct Circle`我们对环的表示包含了对点的表示：

```
struct Circle { const struct Point _; int rad; };
```

但是，我们自然绝冬不去直接的访问者部分。相反，当我们想要继承我们从`Circle`上抛到`Point`并且在这里处理`struct Point`的初始化。

这里有另外一个表示环的方式：它能包含一个点作为一个集合。我们能够仅仅通过指针来处理对象；因此这样的一个环的表示看起来就像如下所示：

```
struct Circle2 {
    struct Point * point;
    int rad;
};
```

这个环一点也不像一个点，也就是说，它不能够从`Point`所继承并且重用它的方法。然而，它能够把点的方法应用到点的部分；它仅仅不能把点的方法用于它自己。

如果一种语言对于继承有明确的符号，差异就会更加明显，相似的表示在C++中会有如下的表示：

```
struct Circle:Point{int rad;};  //inheritance
struct Circle2{ struct Point point;int rad}; //aggregate
```

在C++中作为一个指针我们是不必要访问对象的。

继承，即，从超类来建立子类，而集合，即，把对象的一部分作为另外一个对象的一部分，提供非常相似的功能。这些应用在特殊的设计中通常被所 is-it-or-has-it？的测试所决定：如果一个新类的一个对象仅仅像一些其他类的对象，我们应该使用继承来实现新的类；如果一个新类有一个其他类作为它的状态的一部分对象，我们应该建立集合。

到我们的点所关注的，一个环仅仅是一个大的点，这就是为什么我们使用继承来做一个环的原因。一个方形是一个不明确的例子：我们能通过一个参考点和边的长度来描述它，或我们能够使用端点的对角线或甚至三个角来描述。仅仅带参考点是方形的几分花哨点；其他表示通向集合。　在我们的算术表达式中，我们已经使用了继承从单目到双目操作节点，但是这已经充分的违背了测试。

4.10 多重继承

因为我们使用平凡的标准化C语言。我们不能够隐藏这样的事实——继承意味着在另一个结构的开始包含一个结构体。利用上抛是在子类的对象上重复利用超类方法的关键所在。通过投掷一个结构体起始的地址完成一个从环岛到点的上抛；指针的值并没有改变。

如果我们在其他结构中包含两个及以上的结构体，并且如果我们愿意在上抛期间做一些地址的处理，我们可以称这样的结果为多重继承：一个对象能够像它属于几个类一样了表现。优点似乎是我们不必很仔细的设计继承的关系——我们可以很快的把类仍到一起并且继承我们希望继承的任何东西。缺点是，显然，在我们能够重用方法之前我们得有地址处理机制。

事情能够实际的很快让我们感到迷惑。思考一个文本，一个方形，每一个都有一个继承的引用点。我们能够把他们一起扔到一个按钮上——仅仅存在的问题希望这个按钮应该继承一个或两个引用点。

我们使用标准化C语言拥有很大的优点：它会使这样的事实很明显，即，继承——多重或其他总是伴随着包含而进行。包含，然而也能作为集合被实现。与复杂化语言定义和增加过量实现相比多重继承对于程序员来说要做的更多，这一点也不清晰。我们将使得事情变得简单兵器只做简单的继承。第14章将首要展示多重继承的使用，库的合入能够被集合和消息转换所实现。

第五章 编程经验——符号表

> 来源：https://github.com/oxnz/ooc/blob/master/tex/chapter0x05.tex
>
> 译者：oxnz

一个结构体明智的加长，以此来共享基本结构的功能，可以帮助省去笨重的`union`的使用。特别在具有动态绑定，我们得到了一种统一的且完美健壮的方式处理消息传递。一旦基本机制就位，一个新的扩展类就可以重用基本代码容易的添加。

作为一个例子，我们将会添加关键字、常量、变量和数学函数到第三章开始的小计算器中。所有这些对象存在一个符号表中并且共享相同的名称搜索技术。

扫瞄标识符

在3.2节中我们实现了函数`scan()`，从主程序获取一行输入并在每次调用返回一个输入符号。如果我们想引进关键字，命名常量等等，我们需要扩展`scan()`。像浮点数一样，我们提取字符数字串以供深入分析：

```
#define ALNUM    "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
                "abcdefghijklmnopqrstuvwxyz" \
                "_" "0123456789"

static enum tokens scan (const char * buf) {
    static const char * bp;
    …
    if (isdigit(* bp) || * bp == '.')
        …
    else if (isalpha( *bp) || *bp == '_') {
        char buf[BUFSIZ];
        int len = strspn(bp, ALNUM);

        if (len >= BUFSIZ)
            error("name too long: %-.10s…", bp);

        strncpy(buf, bp, len), buf[len] = '\0', bp += len;
        token = screen(buf);
    }
    ...
```

一旦我们有了一个标识符，我们让新函数`screen()`来决定它的`token`值应当是什么。如果有必要，`screen()`将会存放一个解析器可以识别的符号描述到全局变量`symbol`中。

使用变量

一个变量参与两个操作：它的值被用作一个表达式的操作数，或者表达式的赋值对象。第一中操作是对3.5节中一个简单的`factor()`扩展。

```
static void * factor (void) {
    void * result;
    ...
    switch (token) {
        case VAR:
            result = symbol;
            break;
        ...
```

`VAR`是一个当`screen()`发现适当的标识符的时候放到`token`中的唯一值。有关标识符的附加信息被放到全局变量symbol中。在这种情况下，`symbol`包含一个节点来标示变量作为表达式树中的一个叶子。`screen()`要么找到变量再符号表中或者使用描述`Var`去创建它。

识别一个赋值有点复杂。如果我们的计算器允许如下两种语法的声明，它将会是舒适的：

```
asgn : sum
     | VAR = asgn
```

不幸的是，`VAR`也可以出现在`sum`的左端，也就是说，使用我们递归下降的技术如何识别C风格嵌入赋值不是立即就能清楚的。因为我们想要学到如何操作关键字，我们设置如下的语法：

> 这里有一个技巧：对`sum`做简单尝试。如果返回是下一个输入符号是`=`，`sum`必定是一个变量叶子节点，我们就可以创建这个赋值。

```
stmt  : sum
    | LET VAR = sum
```

这被翻译成如下函数：

```
static void * stmt (void) {
    void * result;

    switch (token) {
        case LET:
            if (scan(0) != VAR)
                error("bad assignment");
            result = symbol;
            if (scan(0) != '=')
                error("expecting =");
            scan(0);
            return new(Assign, result, sum());
        default:
            return sum();
    } /* this i s comet */
}
```

在主程序中我们调用`stmt()`来替代`sum()`，并且我们的识别器已经准备好操作变量。`Assign`是一个新的类型描述，来计算一个`sum`的值并赋值给一个变量。

筛子-- `Name`

赋值具有如下语法：

```
stmt : sum
     | LET VAR = sum
```

`LET` 是关键字的一个例子。在构建筛子的过程中我们仍然可以决定什么标识符将标示 `LET: scan()` 从输入行提取一个标识符并传递给 `screen()` ，是它在符号表中查找并返回 `token` 的适当值，至少一个变量，一个节点在 `symbol` 中。

识别器丢弃 `LET` 但是插入变量作为叶子节点在树中。对于另一个符号，例如一个算数函数的名称，我们可能想要适用 `new()` 到 `screener` 返回的符号来获取一个新的节点。因此，我们的符号表入口应当对与大部分具有相同的函数动态绑定与我们树节点。

对于一个关键字，一个 `Name` 需要包含输入字符串和 `token` 值。稍后我们想要继承 `Name` ；因此，我们定义结构在 `Name.r` 中：

```
struct Name {            /* base structure */
    const void * type;    /* for dynamic linkage */
    const char * name;    /* may be malloc-ed */
    int token;
};
```

我们的符号从不死亡：他们的名字是预定义的常量字符串还是存储的用户自定义变量动态字符串是没有关系的--我们将不会回收他们。

在我们可以定义一个符号之前，我们需要输入它到符号表。这不能通过调用 `new(Name, ...)` 来处理，因为我们想要支持更多比 `Name` 复杂的符号，并且我们想要隐藏符号表的实现。相反的，我们提供一个函数 `install()` ，它需要一个 `Name` 对象并把它插入到符号表中。这里给出符号表接口文件 `Name.h` ：

```
extern void * symbol;    /* -> last Name found by screen() */
void install (const void * symbol);
int screen (const char * name);
```

识别器必须插入像 `LET` 的关键字到符号表中，在他们被 `screener` 发现之前。这些关键字可以被定义进一个常量表结构中--它对 `install()` 没有影响。下面的函数被用来初始化识别：

```
#include "Name.h"
#include "Name.r"

static void initName (void) {
    static const struct Name names [] = {
        { 0, "let", LET },
        0
    };
    const struct Name * np;

    for (np = names; np->name; ++np)
        install(np);
}
```

注意`names[]`，关键字表，不需要被存储。我们适用`Name`的标示来定义`names[]`，也就是说，我们包含`Name.r`。由于关键字`LET`被丢弃，我们不提供动态绑定的方法。

父类的实现-- `Name`

通过名字搜索符号是一个标准问题。不幸的是，ANSI标准没有定义一个合适的库函数来解决它。`bsearch()`--有序表中的二分查找--比较接近，但是如果我们想要插入一个单独的新符号，我们不得不调用`qsort()`来设置阶段给后续搜索。

UNIX系统很可能提供两三个函数家族来处理动态增长表。`lsearch()`--线性搜索一个数组并在`end(!)`添加--不是完全高校的。`hsearch()`--一个结构体哈希表由一个文本和一个信息指针组成--维护一个固定大小的表并且强制一个尴尬的结构体入口。`tsearch()`--一个二叉树具有任意比较和删除--是最常用的家族但是很没有效率，如果初始符号从一个有序序列中安装。

在一个UNIX系统上，`tsearch()`有可能是最好的折衷。对于一个可移植的实现具有二元线程树可以在`[Sch87]`找到。然而，如果这个家族不可用，或者如果我们不能保证一个随机的初始化，我们应当查看一个简单的设备来实现。一个小心实现的`bsearch()`可以很容易的被扩展来支持存储的数组插入：

```
void * binary (const void * key,
    void * _base, size_t * help, size_t width,
    int (* cmp) (const void * key, const void * elt)) {
    size_t nel = * nelp;
#define base    (* (char **) & _base)
    char * lim = base + nel * width, * high;

    if (nel > 0) {
        for (high = lim - width; base <= high; nel >>= 1) {
            char * mid = base + (nel >> 1) * width;
            int c = cmp(key, mid);

            if (c < 0)
                high = mid - width;
            else if (c > 0)
                base = mid + width, --nel;
            else
                return (void *) mid;
        }
```

到这里为止，这是一个任意数组的二元搜索。`key`只想要找的对象；`base`初始值是一个具有`*nelp`个元素的表的开始位置，每个元素`width`字节；并且`cmp`是一个函数用来比较`key`和一个表中的元素。在此我们要么找到一个元素并返回它的位置，要么`base`现在是`key`应当出现在表中的位置地址。我们继续如下：

```
        memmove(base + wdith, base, lim - base);
    }
    ++ * nelp;
    return memcpy(base, key, width);
#undef base
}
```

`memmove()` 移动数组的末尾，`memcpy()` 插入 `key` 。我们假定数组之外还有空间并且我们通过 `nelp` 纪录我们已经加入了一个元素-- `binary()` 和标准函数 `bsearch()` 只需要地址而不是变量的值包含饿了表中元素的个数。

> `memmove()` 复制字节即使源和目标区域重叠；`memcpy()` 并不如此，但是它更具效率

给出一个通用搜索和入口的方式，我们可以很轻易的管理我们的符号表。首先我们需要比较一个key和表中的元素：

```
static int cmp (const void * _key, const void * _elt) {
    const char * const * key = _key;
    const struct Name * const * elt = _elt;

    return strcmp(* key, (* elt) -> name);
}
```

作为一个键值，我们只传递一个指向输入符号文本的指针的地址。表中的元素当然是 `Name` 结构体，并且我们只查看他们的 `.name` 成员。

搜索和入口通过适用适当的参数对 `binary()` 进行调用来实现。由于我们事先不知道符号个数，我们确保一直有空间让我们扩招表：

```
static struct Name ** search (const char ** name) {
    static const struct Name ** names;    /* dynamic table */
    static size_t used, max;

    if (used >= max) {
        names = names
            ? realloc(names, (max *= 2) * sizeof * names)
            : malloc((max = NAMES) * sizeof * names);
        assert(names);
    }
    return binary(name, names, & used, sizeof * names, cmp);
}
```

`NAMES` 是一个定义的常量具有初始分配的表项；每次我们用完，我们都是表的大小加倍。

`search()` 适用指向要查找的文本的地址指针作为参数并且返回表项的地址。如果文本未找到，`binary()` 就插入 `key` --也就是说，只有指向文本的指针，而不是一个 `struct Name` --到表中。这个策略是为了 `screen()` 的利益，它只新建一个表元素，如果一个输入中的标识符是未知的：

```
int screen (const char * name) {
    struct Name ** pp = search(& name);

    if (* pp == (void *) name)    /* entered name */
        * pp = new(Var, name);
    symbol = * pp;
    return (* pp) -> token;
}
```

`screen()` 让 `search()` 查找要显示的输入符号。如果指符号文本的指针被插入符号表，我们需要使用一个新标识符的项目描述替换它。

对于 `screen()` ，一个新的标识符必须是一个变量。我们假定这里有一个类型描述 `Var` 知道如何构建 `Name` 结构体来描述变量并且我们让 `new()` 做剩下的工作。其他的情况，我们让 `symbol` 指向符号表项并且返回它的 `.token` 值。

```
void install (const void * np) {
    const char * name = ((struct Name *) np) -> name;
    struct Name ** pp = search(& name);

    if (* pp != (void *) name)
        error("cannot install name twice: %s", name);
    * pp = (struct Name *) np;
}
```

`install()` 比较简单。我们接受一个 `Name` 对象并且让 `search()` 在符号表中找到它。 `install()` 被假定来只处理新符号，所以我们应当总是能够插入对象替换它的名字。否则，如果 `search()` 真的找到一个符号，我们就有麻烦了。

子类的事先--Var

`screen()` 调用 `new()` 来创建一个新的变量符号并且返回它到识别器，并插入它到一个表达式树中。因此， `Var` 必须创建可以项节点行为的符号表项，也就是说，当定义 `struct Var` 的时候，我们需要扩展一个 `struct Name` 来继承在符号表中存在的能力并且我们必须支持动态绑定的函数可以适用于表达式节点。我们描述接口在 `Var.h` 中:

```
const void * Var;
const void * Assign;
```

一个变量具有一个名字和一个值。如果我们计算一个算术表达式的值，我们需要返回 `.value` 成员。如果我们删除一个表达式，我们一定不能删除变量节点，因为它存活在符号表中：

```
struct Var { struct Name _; double value; };
#define value(tree) (((struct Var *) tree) -> value)

static double doVar (const void * tree) {
    return value(tree);
}

static void freeVar (void * tree) {
}
```

就如在4.6节中讨论的，通过提供一个值的访问函数来简化代码。

创建一个变量需要分配一个 `struct Var` ，插入一个变量名的动态副本，并且标识值 `VAR` 被识别器规定：

```
static void * mkVar (va_list ap) {
    struct Var * node = calloc(1, sizeof(struct Var));
    const char * name = va_arg(ap, const char *);
    size_t len = strlen(name);

    assert(node);
    node-> _.name = malloc(len + 1);
    assert(node -> _.name);
    strcpy((void *) node-> _.name, name);
    node -> _.token = VAR;
    return node;
}

static struct Type _Var = { mkVar, doVar, freeVar };

const void * Var = & _Var;
```

`new()` 照料插入 `Var` 类型描述到节点中，在符号被 `screen()` 返回之前或者任何的使用。

就技术而言， `mkVar()` 是 `Name` 的构建子。然而，只有变量名需要被动态存储。因为饿哦我们决定在我们的计算器中构建子负责分配一个对象，我们不能让 `Var` 构建子调用一个 `Name` 构建子来维护 `.name` 和 `.token` 成员--一个 `Name` 构建子将会分配一个 `struct Name` 而不是一个 `struct Var` 。

赋值

赋值是一个二元操作。识别器保证我们具有一个变量作为做操作数和 `sum` 作为右操作数。因此，我们世纪需要实现的是实际赋值操作，也就是说，动态绑定进类型描述的 `.exec` 成员：

```
#include "value.h"
#include "value.r"

static double doAssign (const void * tree) {
    return value(left(tree)) = exec(right(tree));
}

static struct Type _Assign = { mkBin, doAssign, freeBin };
const void * Assign = & _Assign;
```

我们共享 `Bin` 的构建子和析构子，因此，在算数操作的实现中必须是全局的。我们也共享 `struct Bin` 和访问函数 `left()` 和 `right()` 。所有这些使用 `value.h` 导出并且实现文件 `value.r` 。我们自己的访问函数 `value()` 对于 `struct Var` 故意的允许修改，如此赋值就可以被很优雅的实现。

另一个子类--常量

谁会喜欢输入 `pi` 或者其他数学常量的值呢？我们从Kernighan和Pike's hoc [K&P84]得到线索并且预定义一些常量给我们的计算器。下面的函数需要被调用在初始化识别器期间：

```
void initConst (void) {
    static const struct Var constants [] = {    /* like hoc */
        { &_Var, "PI", CONST, 3.14159265358979323846 },
        ...
        0 };
    const struct Var * vp;

    for (vp = constants; vp -> _.name; ++ vp)
        install(vp);
}
```

变量和常量几乎是一样的：都具有名称和值并且存活在符号表中；都返回他们的值在一个算数表达式的使用中；并且都不应当被删除，当我们删除一个算数表达式的时候。然而，我们不应当给常量赋值，所以我们需要同意一个新的标识符值 `CONST` ，识别器在 `factor()` 中接受就像 `VAR` 一样，但是不允许在 `stmt()` 的赋值的左边。

数学函数--Math

ANSI-C定义了许多数学函数例如 `sin()` , `sqrt()` , `exp()` 等等。作为另一个继承的练习，我们将添加库函数使用一个单个 `double` 参数并且具有一个 `double` 结构到我们的计算器。

这些函数工作的就如同一元运算符一样。我们可以定义一个新的类型给节点给每个函数并且收集大多数功能从 `Minus` 和 `Name` 类，但是这里有一个更简单的方法。我们扩展 `struct Name` 到 `struct Math` 如下：

```
struct Math { struct Name _;
    double (* funct) (double);
};
#define funct(tree) (((struct Math *) left(tree)) -> funct)
```

额外的给函数名称用于输入和标识符给识别，我们存储像 `sin()` 的库函数的地址在符号表项中。

在初始化期间我们调用下面的函数来输入所有的函数描述到符号表中：

```
#include <math.h>

void initMath (void) {
    static const struct Math functions [] = {
        { &_Math, "sqrt", MATH, sqrt },
        ...
        0 };
    const struct Math * mp;

    for (mp = functions; mp -> _.name; ++ mp)
        install(mp);
}
```

一个函数调用是一个因子就好像使用一个减号标记一样。对于识别我们需要扩展我们的语法对因子：

```
factor : NUMBER
      | - factor
      | ...
      | MATH ( sum )
```

`MATH` 是公共标识符对所有函数输入通过 `initMath()` 。这个翻译到下面的附加 `factor()` 在识别器中：

```
static void * factor (void) {
    void * result;
    ...
    switch (token) {
        case MATH:
        {
            const struct Name * fp = symbol;

            if (scan(0) != '(')
                error("expecting (");
            scan(0);
            result = new(Math, fp, sum());
            if (token != ')')
                error("expecting )");
            break;
        }
```

`symbol` 首先包含符号表元素对一个函数例如 `sin()` 。我们保存这个指针并且构建表达式树对于函数参数通过调用 `sum()` 。然后我们使用 `Math` ，类型描述给函数，并且让 `new()` 构建下面的节点给表达式树：

我们让一个二元节点的左边只想符号表元素给函数并且我们附加参数树在右边。这个二元节点具有 `Math` 作为类型描述，也就是说，方法 `doMath()` 和 `freeMath()` 将会被调用来分别执行和删除节点。

Math节点仍然使用 `mkBin()` 构建，因为这个函数不关心指针的后代。 `freeMath()` ，然而，可能只会删除右子树：

```
static void freeMath (void * tree) {
    delete(right(tree));
    free(tree);
}
```

如果我们仔细看上图，我们可以看到一个 `Math` 节点的执行是非常容易的。 `doMath()` 需要调用存储在符号表中元素可以被访问的作为左后代二元节点从之被调用：

```
static double doMath (const void * tree) {
    double result = exec(right(tree));

    errno = 0;
    result = funct(tree)(result);
    if (errno)
        error("error in %s: %s",
            ((struct Math *) left(tree)) -> _.name,
            strerror(errno));
    return result;
}
```

唯一的问题是抓住数字错误通过检测`errno`变量在ANSI-C头文件`errno.h`中声明。这个完成了数学函数的实现给计算器。

总结

机遇一个函数`binary()`来搜索和插入一个有序数组，我们实现了一个符号表包含了就诶够体具有名称和标识符値。继承允许我们插入其他结构体到表中而不需要改变函数搜索和插入。这种方式的高雅变得明显一旦我们考虑一个传统的定义一个符号表元素出于我们的目的：

```
struct {
    const char * name;
    int token;
    union {            /* based on token */
        double value;
        double (* funct) (double);
    } u;
};
```

对于关键字，`union`是没有必要的。用户定义的函数讲会要求一个更详细的描述，并且引用`union`的部分是讨厌的。

继承允许我们适用符号表功能到新的项而不改变已经存在的代码。动态绑定在许多方式帮助保持实现的简单性：常量符号表元素，变量和函数可以被绑定进表达式树中而不用担心我们不慎删除他们；一个执行函数参考自身值有它自己安排节点。

练习

新关键字是必须的用来实现例如`while`或者`repeat`循环，`if`语句等等。识别被`stmt()`处理，但是这个对于大部分情况，只有一个问题编译器构建，而不是继承。一旦我们决定语句描述，我们将创建节点类型例如`While`，`Repeat`或者`IfElse`，并且关键字在符号表中不需要知道他们的存在。

更有趣的是函数具有两个参数的例如`atan2()`在ANSI-C数学库中。从这里看出符号表，这个函数被处理仅仅类似简单函数，但是对于表达式树我们需要开发一个新的节点类型使用三个后代。

用户定义的函数具有一个现实的有意思的问题。如果我们表示单独的参数通过 `$` 并且我们使用一个节点类型 `Parm` 来只想函数项在符号表中从那里我们可以暂时存储参数值只要我们不允许递归，就是简单的。函数具有参数名和几个参数是比较困难的，当然了。然而，这是一个好的练习 `inverstigate` 继承的好处和动态绑定的好处。我们将在第11章中返回到这个问题。